NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

**APPLICATION OF A
MECHANICAL VERIFICATION SYSTEM
TO A
HIGH-SPEED TRANSPORT PROTOCOL**

by

Carl M. Pederson, Jr.

September 1995

Thesis Advisor: Dennis Volpano

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE September 1995	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE
APPLICATION OF A MECHANICAL VERIFICATION SYSTEM TO A HIGH-SPEED TRANSPORT PROTOCOL

5. FUNDING NUMBERS

6. AUTHOR(S)
Pederson, Carl, M., Jr.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The high-speed transport protocol, SNR, has never been completely analyzed. SNR's design incorporates a novel feature, specifically, periodic and frequent exchange of state information to coordinate the actions of the transmitter and receiver. This innovation exploits the higher bandwidth of modern fiber-optic networks to increase data transmission rates.

Traditional methods used to verify SNR have been largely unsuccessful because of the protocol's inherit complexity. The protocol functions as an asynchronous concurrent system and for that reason we apply a mechanical verification tool called Murphi. The Murphi Verification System is used to verify two phases of SNR, the connection establishment phase and data transfer phase operating under Mode 0 (no error or flow control) and Mode 1 (flow control only). The connection establishment phase functions as intended. Murphi detected apparent design flaws in both Mode 0 and Mode 1 of the data transfer phase. Buffer overflow can occur in Mode 1. An unexpected termination of the connection by the receiver is possible in both modes. The feasibility of applying Murphi to verify communication protocols in general is also addressed.

14. SUBJECT TERMS
Verification, transport protocols, SNR, Murphi.

15. NUMBER OF PAGES
128

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

APPLICATION OF A MECHANICAL VERIFICATION SYSTEM TO A HIGH-SPEED TRANSPORT PROTOCOL

Carl M. Pederson, Jr.
Commander, United States Navy
B.A., Whitman College, 1977

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

September 1995

Author: [illegible]
Carl M. Pederson, Jr.

Approved by: Dennis Volpano
Dennis Volpano, Thesis Advisor

[illegible]
Gilbert M. Lundy, Second Reader

Ted Lewis
Ted Lewis, Chairman,
Department of Computer Science

ABSTRACT

The high-speed transport protocol, SNR, has never been completely analyzed. SNR's design incorporates a novel feature, specifically, periodic and frequent exchange of state information to coordinate the actions of the transmitter and receiver. This innovation exploits the higher bandwidth of modern fiber-optic networks to increase data transmission rates.

Traditional methods used to verify SNR have been largely unsuccessful because of the protocol's inherit complexity. The protocol functions as an asynchronous concurrent system and for that reason we apply a mechanical verification tool called Murphi. The Murphi Verification System is used to verify two phases of SNR, the connection establishment phase and data transfer phase operating under Mode 0 (no error or flow control) and Mode 1 (flow control only). The connection establishment phase functions as intended. Murphi detected apparent design flaws in both Mode 0 and Mode 1 of the data transfer phase. Buffer overflow can occur in Mode 1. An unexpected termination of the connection by the receiver is possible in both modes. The feasibility of applying Murphi to verify communication protocols in general is also addressed.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

I. INTRODUCTION

A. BACKGROUND

When building a program or system, proper operation of the entity is desired. However, often it does not behave as expected. The improper behavior may be the result of a flawed conceptual design used as the basis for the implementation. Detecting and eliminating errors in the design and implementation of a program or system greatly enhances the likelihood it will function correctly. It is very difficult to assure a non-trivial program is free of logical errors. Concurrent systems -- such as a communication protocol -- turn out to be some of the most complex programs.

Checking the correctness of a concurrent program is usually extremely challenging. Manual analysis methods are often inadequate because of the inherent complexity. Testing techniques, such as simulation, fall short because of the difficulty of exercising all possible interactions in the context of nondeterministic execution. Computer aided verification techniques and tools have been developed to address the problem. One such automatic tool is the Murphi Verification System developed by D. Dill et al. [DDHY92].

The Murphi Verification System allows the user to specify properties for a finite state asynchronous concurrent system and then check whether they are violated by the system. The properties to be checked, the initial conditions, and allowable state transitions of the system being verified are written in the Murphi Description Language. The Murphi Compiler is then used to produce an executable program that will: 1) generate all system states, 2) check the invariance of the designated properties in each of these states, and 3) report violations of correctness properties. Verifiable properties include the absence of deadlock, mutual exclusion, and others specified by the tool user which are considered important to the desired behavior of the concurrent system being examined.

This thesis verifies the design of a non-trivial concurrent system, specifically, the high speed transport protocol SNR [NRS90]. SNR may play a significant role in the context of very high-bandwidth communications made possible by optical fiber. SNR's design incorporates a novel feature, specifically, periodic and frequent exchange of state information to coordinate the actions of the transmitter and receiver. This innovation exploits the higher bandwidth of modern networks to increase data transmission rates.

The essential properties of SNR have not been verified, or for that matter even adequately formalized. Attempts to verify that the protocol is free of logical errors have produced only limited results [McAr92], [Tipi93]. The analysis of SNR, described in these two documents, was conducted without first rigorously asserting specific properties being examined. A more structured approach, with verification as the primary goal, should to be taken. There is without question, a need to begin identifying key properties, formally describing them, and finally verifying that SNR has these properties.

B. OBJECTIVES

Because of the complexity of communication protocols, formal verification of a protocol's design is typically not attempted. Instead testing techniques such as simulation are often used to determine if the protocol will operate as expected. A relatively mature mechanical verification system has yet to be applied to SNR.

The primary objectives of this thesis are:

- identify and develop formal specifications of key properties of SNR, and
- verify these properties using the Murphi Verification System.

A secondary objective is to explore the feasibility of applying a mechanical verification tool, such as Murphi, to communication protocols.

C. THE RESEARCH QUESTION

This thesis will attempt to answer the following specific questions:

- What particular properties must SNR exhibit to ensure proper behavior when used as the transport layer for a high speed communication network?
- Is SNR's behavior consistent with these desired properties?
- What properties of the protocol can be checked with Murphi?

- Are there properties that can not be checked? If so, what limitations in the tool prevent their verification?
- State-space explosion is likely to be encountered. Can state-space reduction methods be employed to overcome the problem, if it occurs?
- What advantages and disadvantages are inherent to the application of automatic verifiers to protocols?

D. SCOPE, METHODOLOGY, AND LIMITATIONS

This thesis examines the high speed transport protocol, SNR as presented in [NRS90]. No attempt is made to improve upon or redesign the protocol. The focus is verifying SNR's design, not discussing the strengths or weakness of a specific protocol.

Since success in the application of Murphi to SNR is uncertain, the verification is conducted in stages. The least complex aspects of SNR are examined first. This approach facilitates the early identification of potential "show stoppers" and allows work done in the initial steps to serve as a foundation for the later and more complicated stages.

Inconsistencies in the original specification of SNR and state space explosion prevented exhaustive verification of SNR. Verification is limited to the connection establishment phase of SNR, and two of the three operating modes of SNR's data transfer phase (Mode 0 -- no error or flow control and Mode 1 -- flow control only). SNR's data transfer phase operating in Mode 2 (both flow control and error enabled) is not verified.

The verification effort of SNR would be greatly enhanced by either 1) the existence of a single source accurate specification for SNR, or 2) coupling verification with a redesign effort so that problems discovered during verification could be address as part of the design process.

E. RELATED WORK

Previous work on SNR, [McAr92] and [Tipi93], provides comprehensive specifications of SNR and alternative interpretations of some of its design objectives. Although these documents are not the primary reference for the thesis, they aided significantly the translation of SNR into Murphi's Descriptive Language. The effort in [McAr92], focused mainly on obtaining an accurate specification for SNR and on analysis of its functional efficiency. Further refinements to the specification and an attempt to examine the behavior of SNR more deeply was made in [Tipi93] and [LuTi94].

A software implementation of the SNR's transmitter portion and receiver portion is presented in [Mez95] and [Wan95] respectively. Test results of this implementation are given in [Gri95].

F. ORGANIZATION

Concurrent systems and their basic properties are discussed in Chapter II. First concepts fundamental to concurrent systems are introduce and then examples are used to illustrate a few of the central ideas.

In Chapter III, concurrent system verification methods are discussed. Mechanical verification and manual proof methods are demonstrated. A brief introduction to the concept of state space explosion and state space reduction techniques is also provided. Finally, verification concerns specific to protocols are addressed.

Chapter IV covers the Murphi Verification System. Its key features are explained.

The SNR protocol is described in Chapter V. An overview of the protocol is given followed by a detailed treatment of its organization. The operation of SNR's data transfer phase is also explained.

The verification of SNR's connection establishment phase and its data transfer phase (operating in Mode 0 and Mode 1) is presented in Chapter VI and Chapter VII, respectively. Each chapter covers the key properties of the phase being examined and the Murphi description used for its verification. Implementation problems encountered and inconsistencies uncovered in SNR's specification are also discussed.

Conclusions and recommendations are provided in Chapter VIII.

II. CONCURRENCY CONCEPTS

A. FUNDAMENTALS

Many practical situations involve concurrent systems and related concepts. For example, the preparation of a meal often allows performing tasks in parallel -- two or more dishes can be cooking at the same time. Typically, communication protocols execute as a concurrent system. Another example, familiar to all computer users, is a computer's operating system. While concurrent systems are frequently encountered, formal terms and analysis methods relevant to concurrency are often unfamiliar. This chapter introduces basic concepts of concurrent systems. These concepts are then revisited in the next chapter in the context of verification.

1. General

In [Ben93], a concurrent program is defined as follows, "A *concurrent program* consists of a set of program fragments called *processes* that can potentially be executed in parallel." This definition suggests characterizing a *concurrent system* as consisting of individual *entities* that operate in parallel. These distinct modules could be programs, machines, or any other types of agents that perform a process. A concurrent system may use some method of synchronization or communication to coordinate the actions of the separately running units. Typically, global variables shared by the individual units are used for the exchange of information. Message passing is another method employed to facilitate cooperation among the separate entities. When explicit synchronization is not part of the system and the separate processes can run at arbitrary speeds, the concurrent system is referred to as an *asynchronous concurrent system*.

It should be clear that a concurrent program is a type of concurrent system. Although only concurrent programs are used in the examples in this chapter, the ideas discussed below apply to all types of current systems, even a concurrent system whose implementation may include hardware in addition to software.

2. States and Transitions

The set of allowable states (or reachable states) consist of those states which can be reached via an execution path. The initial system state is refereed to as the *start state*. There also may be states that can only be entered if an error occurs in the system. The global system state (or *global state*) is defined by the values of variables comprising the concurrent system, both global variables and variables local to the separate units. System variables may by explicit or concealed. A program counter, local to a process, is an example of a hidden variable. The size of the global state-space is determined by the domain of each variable. A global state can be represented by a n-tuple, with each component representing one of the system variables. If a system consists of n variables, and each variable can take on a value from domain D_i, then D_1 x D_2 x ... x D_n defines the total size of the system state space. The number of states available to the system can be very large. It is typical, in real world systems, for the number of reachable state to be so large as to preclude exhaustive system analysis. When this situation occurs it is referred to as "state space explosion".

When and how the values of system variables can change, characterize the *transitions* allowed in a concurrent system. Each process of a concurrent system can perform *atomic actions*. These actions define the system transitions. Once started an atomic action is executed indivisibly until complete. Individual instructions that may comprise the action are assumed to execute instantly. The *granularity* of the atomic actions depends on the level of *abstraction* used for a particular representation. The level of abstraction, in turn, determines the kind of analysis that can be performed.

An important aspect of any concurrent system is the *interleaving* of its atomic actions. Under many conditions the execution ordering, among the atomic actions of the various modules comprising the concurrent system, is nondeterministic. This greatly increases the complexity encountered when considering concurrent systems.

The history of a concurrent program's transitions can either be described using a sequence of states or working from the start state and applying the sequence of atomic actions executed. Either history can be used to generate the other. The sequence of atomic actions can be reconstructed from a listing of states by noting the change in the program counter from one state to the next. From a sequence of transitions the sequence of states is obtained by simulating the actions. Each history corresponds to one possible interleaving for the system. The set of all histories characterizes completely the behavior of the concurrent system. Correct behavior of the system depends on whether the set of all possible interleavings exhibits certain properties.

3. Properties

Various properties can be attributed to a concurrent system. Properties commonly belong in one of two categorizes, **safety** or **liveness**. A liveness property asserts progress will be made by a program. A safety property says that, if a program makes progress it does so without error. Some general properties are *deadlock, fairness (starvation),* and *livelock*. For concurrent systems, deadlock has occurred when no other states can be reached other than the current state. Livelock is similar to deadlock, but with some number of transitions occurring (i.e., altering the global state) before the current state is repeated and where no overall progress takes place. Fairness (or absence of starvation) is the condition that if a module is ready to perform an action it will be given the opportunity to carry out that action (no appropriate action is indefinitely delayed).

B. EXAMPLES

To illustrate concepts and properties discussed above, four programs that attempt to implement the familiar notion of mutual exclusion, are analyzed. These examples are based on material from [Ben90] and [Ben93]. The programs introduced below will again be used in Chapter IV to demonstrate some of the features and behavior of Murphi.

1. Program Description

The programs used in the examples consist of two processes, each executing a loop containing a sequence of instructions. The two processes are referred to as P1 and P2. The statements are presented using an Ada-style syntax. Each process contains a non-critical section, a critical section, some means to signal the other process when it is in its critical section (global variables C1 and C2), and a method to control critical section entry. A process may stay in its non-critical section indefinitely. When ready, a process requests entry to its critical section and waits until admitted. Once in its critical section the process will eventually exit. The general pattern is shown below.

```
subtype TEST_VAR_TYPE is integer range 0..1;
C1, C2: TEST_VAR := 1;   -- global

    Process P1                              Process P2
        loop                                    loop

        L1.1   non_critical_section;            L2.1   non_critical_section;
        L1.2   entry_request_section;           L2.2   entry_request_section;
        L1.3   critical_section;                L2.3   critical_section;
        L1.4   post_critical_section;           L2.4   post_critical_section;

        end loop;                               end loop;
```

Important locations in each process are labeled. For purposes of the thesis, these labels function as program counters. The first label in P1 is *L1.1*. It indicates P1 will next execute the statement(s) between *L1.1* and *L1.2*. (In a single processor system, this will be the next time it is P1's turn to run.) Similarly the first statement in P2 is *L2.1*, second *L2.2*, etc.

The system consists of four variables, two global variables -- C1 and C2, and two local variables -- the program labels for P1 and P2. Specifying their values defines a **global state** of the system. An example of a global state: P1 at *L1.1* (or in a short hand notion, *L1.1*), C1 = 1, P2 at *L1.1*, and C2 = 1. Written as a 4-tuple, (L1.1, 1, L2.1, 1). The domain of C1 and C2 is $\{0,1\}$. The domain of locations, for P1 is $\{L1.1, L1.2, L1.3, L1.4\}$, likewise for P2. The size of the global space state is

$$|C1| \times |C2| \times |P1\_Locations| \times |P2\_Locations| = 2 \times 2 \times 4 \times 4 = 64.$$

A process's action may cause the value of a variable (or variables) to change. For example, (L1.**1**, 1, L2.3, 1) $\rightarrow$ (L1.**2**, 1, L2.3, 1) is a state transition where the program counter for P1 went from *L1.1* to *L1.2*. Note changing the program counter for a process is considered an atomic action even when multiple processor instructions are involved.

The entry_request_section and post_critical_section are instantiated with specific instructions for each example presented below in Sections 3, 4, 5, and 6. The sequence of statements from one example to the next change only slightly but the mutual exclusion algorithm's behavior is usually significantly altered.

2. Program Properties

The desired properties for the example programs attempting to provide mutual exclusion are:

Category	Property
Safety	1. Mutual Exclusion - the two processes can not be in their critical regions at the same time.
Liveness	2. Fairness - if a process seeks to enter its critical region, eventually it will be allowed. (Or stated differently -- all execution interleavings allow a process to eventually enter its critical section if the process seeks entry). 3. Absence of Deadlock -- no execution interleaving exists that results in both processes attempting to enter their critical region, but neither can succeed. 4. Absence of Livelock -- no execution interleaving exists, which **could continue indefinitely,** that does not allow either process to enter its critical section.

Table 1. Desired Properties for Mutual Exclusion Algorithms.

The first three programs used in the examples violate one or more of the above properties. In contrast to the other examples, Peterson's Algorithm satisfactorily achieves mutual exclusion and the safety property without violating the liveness properties.

3. Example 1 -- Mutual Exclusion Violation

The program listed below operates as follows. When P1 desires to enter its critical section, it tests C2 to determine if it can safely enter. If C2 = 0, P1 waits in the inner loop until P2 has departed its critical section and sets C2 to 1. When C2 = 1, P1 sets C1 to 0 and enters its critical section. After P1 exits its critical section it sets C1 to 0. Process P2 is analogous to P1.

```
subtype TEST_VAR_TYPE is integer range 0..1;
C1, C2: TEST_VAR := 1;  -- global, set to 0 inside critical section, set to 1 after exiting

task body P1 is                                task body P2 is
begin                                          begin
        loop                                           loop

L1.1    Non_Critical_Section_1;                L2.1    Non_Critical_Section_2;
L1.2    loop  exit when C2 = 1;   end loop;    L2.2    loop  exit when C1 = 1;   end loop;
L1.3    C1  :=  0;                             L2.3    C2  := 0;
L1.4    Critical_Section_1;                    L2.4    Critical_Section_2;
L1.5    C1  :=  1;                             L2.5    C2  :=  1;

        end loop;                                      end loop;
end P1;                                        end P2;
```

This program violates mutual exclusion (MUEX). The following interleaving results in P1 and P2 in their critical sections simultaneous: *L1.1, L2.1, L1.2, L2.2, L1.3, L2.3, L1.4, L2.4*. From the starting condition, (L1.1, 1, L2.1, 1), alternate execution of statements in each process can set up an unsafe situation. The system can find itself with P1 waiting at *L1.3*, after testing C2 and exiting the inner loop but prior to setting C1 to 0, when P2, with its program counter at *L2.2*, gets its turn to run. P2 test C1, finds it equals 1, and exits its inner loop. P2's program counter changes to *L2.3*. At this point, executing statements from each process alternately results in the violation. Figure 1 illustrates the transitions leading to the violation of MUEX.

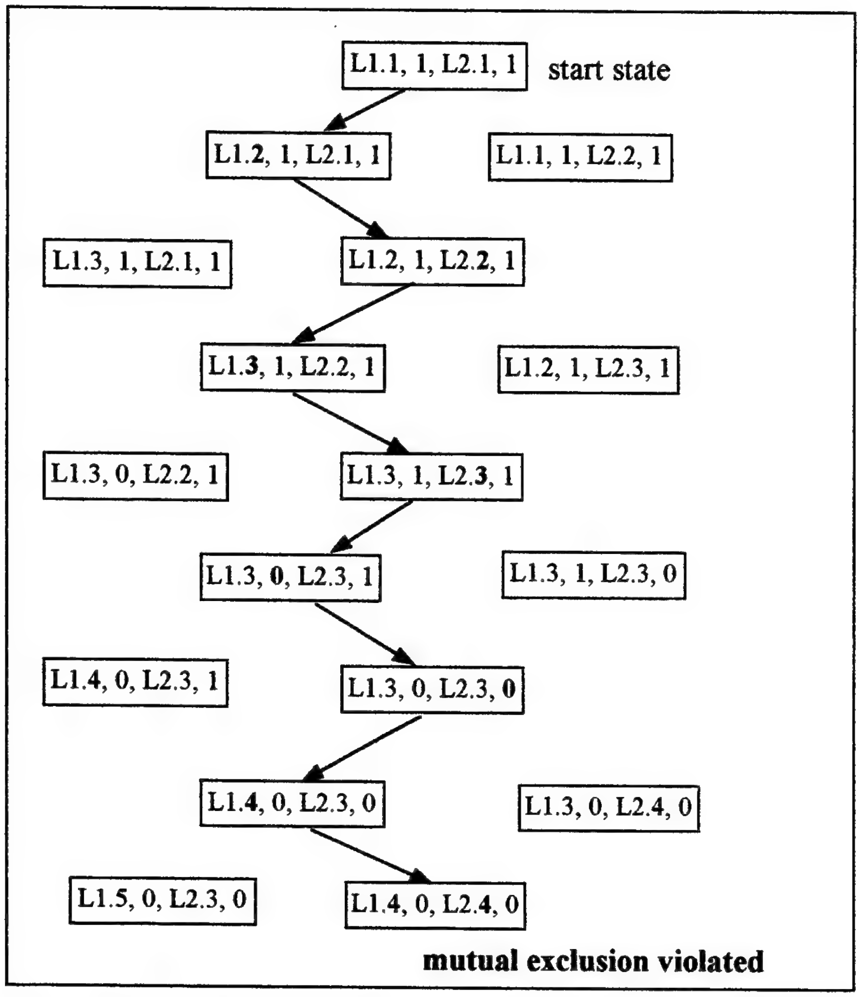

Figure 1. State Transitions Leading to MUEX Violation.[1]

4. Example 2 -- Deadlock

Unlike the first example this program satisfies mutual exclusion. When P1 seeks critical section entry, it sets C1 to 0 and then tests C2 to determine if can enter. If C2 = 0, P1 waits in a loop until P2 has set C2 to 1. When C2 = 1, P1 enters its critical section. After P1 leaves its critical section C1 is set to 0. Process P2 works the same as P1.

[1]State labels have the following format: P1's program location, C1's value, P2's program location, C2's value. Only those states immediately available from the pervious state are shown. The path leading to the violation of MUEX is indicated by arrows.

```
subtype TEST_VAR_TYPE is integer range 0..1;
C1, C2: TEST_VAR := 1;   -- global, set to 0 when process wishes to enter critical
section,
                                -- set to 1 after exiting

task body P1 is                              task body P2 is
begin                                        begin
        loop                                         loop

L1.1    Non_Critical_Section_1;              L2.1    Non_Critical_Section_2;
L1.2    C1  := 0;                            L2.2    C2  := 0;
L1.3    loop  exit when C2 = 1;   end loop;  L2.3    loop  exit when C1 = 1;   end loop;
L1.4    Critical_Section_1;                  L2.4    Critical_Section_2;
L1.5    C1  := 1;                            L2.5    C2  := 1;

        end loop;                                    end loop;
end P1;                                      end P2;
```

Although the program accomplishes MUEX, it exhibits deadlock. The following interleaving results in a situation with P1 and P2 unable to make further progress: *L1.1, L2.1, L1.2, L2.2, L1.3, L2.3.* At *L1.2* P1 signals its intention to enter its critical section and then stops at *L1.3,* while P2 executes. P2 does the same thing. P1 sets C1 and P2 sets C2 just before the other process checks if entry is allowed. Both P1 and P2 become "stuck" in their inner loops unable to exit. The global state remains at (L1.3, 0, L2.3,0) since the action of the inner loop does not change the value of any system variables. Deadlock occurs in this program when atomic actions from each process execute alternately. Interleavings that do not alternately set C1 and C2 to 0 do not violation this liveness property.

5. Example 3 -- Starvation and Livelock

The program listed below works as follows. When P1 desires to enter its critical section, it sets C1 to 0 and then tests C2 to determine if it can safely enter. If C2 = 0, P1 gives ups its attempt to enter its critical region by setting C1 to 1. P1 then resets C1 to 0, and tries once again to enter its critical region. When C2 = 1, P1 exits the inner loop and enters its critical section. After P1 completes its critical section it sets C1 to 0. Process P2 works just like P1. Because of the concurrent nature of this program, the two statements inside the inner loop, where C1 is assigned one value and then another value, are important to the behavior of the algorithm. Each assignment is atomic and the resulting state transition may enable an action in the other parallel process.

```
subtype TEST_VAR_TYPE is integer range 0..1;
C1, C2: TEST_VAR := 1;  -- global, set to 0 when process wishes to enter critical
section,
                             -- set to 1 after exiting

task body P1 is                          task body P2 is
begin                                    begin
        loop                                     loop

L1.1    Non_Critical_Section_1;          L2.1    Non_Critical_Section_2;
L1.2    C1  :=  0;                       L2.2    C2  := 0;
L1.3    loop  exit when C2 = 1;          L2.3    loop  exit when C1 = 1;
L1.4        C1  :=  1;                   L2.4        C2  :=  1;
L1.5        C1  :=  0;                   L2.5        C2  :=  0;
        end loop;                                end loop;
L1.6    Critical_Section_1;              L2.6    Critical_Section_2;
L1.7    C1  :=  1;                       L2.7    C2  :=  1;

        end loop;                                end loop;
end P1;                                  end P2;
```

Starvation and livelock are both possible in this program. Explained first is the circumstances leading to starvation, then livelock is covered.

Starvation occurs under the following situation: P1 at *L1.2* and P2 at *L2.2*; P2 transitions to *L2.3* and checks C1. Since C1 = 0, P2 cannot exit the inner loop and enter its critical region so P2 transitions to *L2.4* setting C2 an to 1 (Note since arbitrary interleaving is possible, P2 could have stayed at *L2.3* and P1 transition to *L1.3*); P1 transitions to *L1.3*, checks C2 and advances to *L1.6*, entering its critical section; P1 can now exit its critical section, set C1 to 1, execute its non-critical section, set C1 to 0 and end up back at *L1.2*. This particular sequence can continue without P2 getting an opportunity to enter its critical section. Likewise, starvation of P1 can occur.

The following interleaving illustrates a situation where P1 and P2 continue to execute a sequence indefinitely (livelock) :

L1.2, L2.2, L1.3, L2.3. L1.4, L2.4, L1.5, L2.5,
L1.3, L2.3, L1.4, L2.4, L1.5, L2.5,
L1.3, L2.3, L1.4, L2.4. L1.5, L2.5 ...

In this particular interleaving, neither P1 nor P2 enters its critical section. This occurs because the setting and resetting of the global variables C1 and C2 is not coordinated with the test of the loop exit condition. Any deviation in the sequence will break the livelock and allow progress.

6. Example 4 -- Peterson's Algorithm

Peterson's Algorithm, as implemented below, is similar to the program given in Section II.B.4 (example 2 above), except the addition of the global variable LAST prevents violation of desired program properties. LAST indicates which process, P1 or P2, most recently executed its critical section. If both processes request entry to their critical section at the same time, LAST is used to break the tie. The process that has waited the longest is allowed to enter. The global variables C1 and C2 are used, as before, to indicate a process's desire to enter its critical section.

```
subtype TEST_VAR_TYPE is integer range 0..1;
subtype LAST_TYPE is integer range 1..2;
C1, C2: TEST_VAR := 1;     -- global, set to 0 when process wishes to enter its critical
                           -- section, 1 after exiting
LAST: LAST_TYPE := 1;      -- global, indicates most recent process to execute its critical
                           -- section, used to break ties
```

```
task body P1 is
begin
        loop

L1.1    Non_Critical_Section_1;
L1.2    C1 := 0;
L1.3    LAST := 1;
L1.4      loop
            exit when C2 = 1 or LAST /= 1;
          end loop;
L1.5     Critical_Section_1;
L1.6     C1 := 1;

        end loop;
end P1;
```

```
task body P2 is
begin
        loop

L2.1    Non_Critical_Section_2;
L2.2    C2 := 0;
L2.3    LAST := 2;
L2.4      loop
            exit when C1 = 1 or LAST /= 2;
          end loop;
L2.6    Critical_Section_2;
L2.7    C2 := 1;

        end loop;
end P2;
```

The safety property, mutual exclusion, and the liveness properties are satisfied by Peterson's Algorithm. To gain some insight into its behavior let's compare the programs from example 2 and 3 with Peterson's Algorithm. In example 2, deadlock occurred when atomic actions from each process were executed alternately. Both P1 and P2 became "stuck" in their inner loops when each process signaled their intentions to enter its critical section just before the other process checked if entry was allowed. With a similar alternating execution interleaving, deadlock is avoided in Peterson's Algorithm because both P1 and P2 set the variable LAST. Suppose P1 is at *L1.4* and P2 is at *L2.4* and P2 had just assigned LAST the value 2. Assume now P1 starts to execute again, and the loop condition at *L1.4* is checked. Since LAST = 2, P1 can exit its inner loop and enter its critical section. Starvation and livelock as demonstrated in example 3 is avoided in Peterson's Algorithm. The assignment made to LAST in each process prohibit that process from starving the other process. Analogously, since an execution sequence can not be immediately repeated, livelock is also prevented.

C. COMMUNICATION PROTOCOLS

Communication protocols are often implemented as communicating concurrent programs. Thus, like all concurrent systems protocols can be ascribed properties. Correct information transfer is the desired primary property of any protocol. The liveness properties covered above also apply and safety properties specific to a particular protocol can be formulated. Some examples include:

- Safety -- The number of messages acknowledged by the receiver is the same as the number sent by the transmitter.
- Safety -- If any data is delivered to the destination it is the same as the data given to the protocol by the source.
- Liveness -- If the transmitter's host has a message to send it is eventually delivered to the receiver's host.

Another interesting concept applicable to protocols (and other concurrent systems) is *self stabilization* as discussed in [GoMu91]. A concurrent system is usually designed so that only safe states are reachable from the start states of the system and program execution from any safe state results in another safe state. Under normal execution, transitions to unsafe states are not allowed. A system is self stabilizing if a system in an **unsafe** state can reach a **safe** state after completing a finite number of actions. The system could have ended up in an unsafe state as a result of any number of causes. A communication protocol could find itself in an unsafe state by such actions as, improper initialization, the corruption of a packet's sequence number, the transmitter or receiver 'crashes' and then recovered, etc.

III. VERIFICATION OF CONCURRENT SYSTEMS

A. METHODS

Verification, in the framework of software engineering, entails checking if the program meets its specification. In the context of this paper, verification involves characterizing the concurrent system in some language, deductive system or modeling scheme and then showing that the behavior of this description satisfies the correctness criteria given in the specification. One method of specifying the correctness criteria for a system is to list required program properties. The task then is to show these properties remain true in all reachable states. This chapter introduces formal methods for verifying concurrent systems.

Various approaches for modeling and verifying concurrent systems have been employed. Finite-state modeling methods, such as communicating finite state machines (CFSMs), Petri nets, and Kripke structures have been used to represent concurrent systems. The model chosen usually depends on the type of analysis being performed and the behavior exhibited by the system. Concurrent system verification includes such activities as:

- proof construction using axioms and inference rules of a logic, or
- analyzing the set of possible system states.

Each approach has particular advantages and disadvantages depending on the concurrent system being examined.

1. Formal Proofs in Logical Systems

Formal proofs are the most familiar approach used for concurrent system verification. Mathematical proofs are constructed, showing the truth of system properties expressed as propositions. Various types of logical systems have been used, including temporal logic. A proof constructed manually may require considerable ingenuity to manageably organize the proof. The process can be quite tedious, and, due to limitations on human's abilities to deal with complex system, is prone to errors. Mechanical theorem provers have not provided as much help as hoped in constructing mathematical proofs for concurrent systems. The task of proving the correctness of programs described in even some of the more simple deductive logical systems is inherently difficulty. To illustrate the manual proof procedure, a formal proof for the mutual exclusion algorithm of example 2 (from Section II.B.4) is given below.

a. Formal Proof Example

This section illustrates the formal proof verification approach. It uses the mutual exclusion algorithm presented in Section II.B.4. This algorithm satisfies the mutual exclusion property but can deadlock. To prove mutual exclusion it must be shown that this property is satisfied in all allowed states. A proof of mutual exclusion is given below using propositional logic.

The program is reproduced in this section for easy reference.

```
subtype TEST_VAR_TYPE is integer range 0..1;
C1, C2: TEST_VAR := 1

task body P1 is                              task body P2 is
begin                                        begin
        loop                                         loop

L1.1    Non_Critical_Section_1;              L2.1    Non_Critical_Section_2;
L1.2    C1  :=  0;                           L2.2    C2  := 0;
L1.3    loop  exit when C2 = 1;   end loop;  L2.3    loop  exit when C1 = 1;   end loop;
L1.4    Critical_Section_1;                  L2.4    Critical_Section_2;
L1.5    C1  :=  1;                           L2.5    C2  :=  1;

        end loop;                                    end loop;
end P1;                                      end P2;
```

The mutual exclusion property is false if P1 is at *L1.4* and P2 is at *L2.4* at the same time. Expressed as a logic formula, $\neg$ (**at**(*L1.4*) $\vee$ **at**(*L2.4*)). This formula must be shown to be invariant through all transitions. The proof is based on induction on the execution sequence and was originally presented in [Ben90]. It is reproduced here using notation consistent with that used in this thesis.

First two lemmas are required for the proof.

Lemma 1. $C1 = 0 \equiv$ **at**(*L1.3*) $\vee$ **at**(*L1.4*) $\vee$ **at**(*L1.5*) is invariant.

This states, when P1 is at *L1.3* or *L1.4* or *L1.5* the variable C1 is 0 and when C1 is 0, P1 is at *L1.3* or *L1.4* or *L1.5*.

Proof of Lemma 1.

Basis Step: C1 is initialized to 1 and the first statement label in P1 is *L1.1*. Thus both sides of the equivalence are false, so the formula of Lemma 1 is true prior to executing any of the statements in the program.

Inductive Step: Assume the formula is true before any statement is executed. We need to show that the formula is true after each statement is executed. All possible transition of P1 and P2 will be examined.

1. *L1.1* to *L1.2* : The truth of the formula is not affected by this transition. If the formula was true before making the transition, it is true after the transition.
2. *L1.2* to *L1.3*: C1 is assigned 0, making the left side of the formula true. The process advances to *L1.3*, making the right side true since P1 is now at *L1.3*.
3. *L1.3* to *L1.3*: For this transition to take place, the test of $C2 = 1$ must be false. The loop is not exited. The truth of the formula is not affected by this transition since no assignment to C1 occurred and P1 remained at *L1.3*.
4. *L1.3* to *L1.4*: The test of $C2 = 1$ must be true for this transition, however there is no affect on the formula.
5. *L1.4* to *L1.5*: Again no affect on the formula.
6. *L1.5* to *L1.1*: C1 is assigned 1 and the program location for P1 changes to *L1.1*. The left side of the formula is now false. Also the right side becomes false since P1 is not at *L1.3*, *L1.4* or *L1.5*. The equivalence remains true.
7. Transitions in P2: The formula is not affected since assignment to C1 does not occur in P2, and the execution location in P1 is not changed by P2.

Hence the formula of Lemma 1 is invariant since it is initially true and remains true through all transitions. $\therefore$

Lemma 2. $C2 = 0 \equiv$ **at**(*L2.3*) $\vee$ **at**(*L2.4*) $\vee$ **at**(*L2.5*) is invariant.

Proof of Lemma 2.

A symmetric proof of the invariance of the formula of Lemma 2 follows from the proof above for Lemma 1.

Theorem 1. The formula $\neg$ (**at**(*L1.4*) $\wedge$ **at**(*L2.4*)) is invariant.

Proof of Theorem 1.

Initially P1 and P2 are at *L1.1* and *L2.1*, respectively. Each proposition is false, so the conjunction is false, and the negation makes the formula true. Thus the formula is initially true.

The only transitions that can affect the truth of the formula are *L2.3* to *L2.4* in P2 while **at**(*L1.4*) or advancing from *L1.3* to *L1.4* in P1 while **at**(*L2.4*). By Lemma 1, **at**(*L1.4*) implies $C1 = 0$, so the transition *L2.3* to *L2.4* can not occur since $C1 = 1$ must be true for P2 to exit the loop and make the transition. By Lemma 2 **at**(*L2.4*) implies $C2 = 0$, so the transition *L1.3* to *L1.4* is impossible for P1.

The formula is not falsified by any of the program's possible transitions, thus the formula is always true. $\therefore$

(The proof can be simplified by using a *proof by contradiction* as follows. Suppose (**at**(*L1.4*) $\wedge$ **at**(*L2.4*)) is true. Then $C2 = 1$ or $C1 = 1$ which contradicts Lemmas 1 and 2. Therefore $\neg$ (**at**(*L1.4*) $\wedge$ **at**(*L2.4*)) must always be true.)

It is interesting to note that a different proof of this algorithm is given in [Ben82]. Its approach is to show that the proposition, **(P1 in its critical section)**, implies the proposition, **(P2 is not in its critical section)**. Similarly, multiple formulations of correctness proofs for the alternating bit protocol can also be found in the research literature on concurrent system verification. They are usually constructed to demonstrate some particular proof technique or system.

2. State Enumeration and Analysis

Verification based on state enumeration usually proceeds by describing the concurrent system in a particular model. Transition relations and properties of the concurrent system are described at the level of detail appropriate for the desired analysis. The description can be in such forms as finite-state machines, programming-language like notion, a Petri net, or a finite Kripke structure. The system states are then generated based on this description. Each generated state is examined to determine if it is consistent with the specification. The specification includes properties required to be true in all reachable states. If an undesirable state can be reached then the concurrent system's design has been shown to contain an error.

The main advantage of using state enumeration and analysis for verification is it can be automated. The Murphi Verification System and the SMV model checker [McMi92] are examples of automatic verification tools. Verifying a concurrent system with Murphi involves describing its behavior in the descriptive language recognized by Murphi, generating a C++ program from the Murphi description, and then running the program to check the invariance of desired properties in all reachable states. See Chapter IV for a detailed discussion of the Murphi Verification System. Section IV.D. presents the Murphi descriptions for the same mutual exclusion algorithm used in the manual proof example.

The process of describing the actions and properties of a concurrent system for a mechanical verification system has its disadvantages. The difficulty is assuring the description will generated **all** allowable states. Will some interleaving be omitted because of an error in the description? Also producing correct descriptions of desired properties can be more problematic than it appears. It is not always easy to translate a simple property expressed in natural language into the formal language used by the verifier.

Another significant disadvantage is the problem of state-space explosion. An exponential growth in the size of possible system states occurs as the complexity of the system increases. Verification may not always be feasible, because of time and space constraints, without employing space state reduction techniques. The next section discusses methods to lessen the state space explosion problem.

B. STATE SPACE REDUCTION TECHNIQUES

The size of a concurrent system's state space may grow very rapidly as the domain or number of system variables increases. Employing state-space reduction techniques, such as eliminating redundant interleaving, folding related states, and down-scaling the concurrent system under analysis, can help. The basic idea is to develop an approximation of the concurrent system. The approximation is achieved by exploiting characteristics such as *symmetry* or *equivalence classes* inherit in the structure of the concurrent system. The trick is to eliminate states without loss of analysis precision by ignoring some level of detail (increasing the level of abstraction). Significant space state reduction has been achieved by the application of these methods. A reduction of over 90% is reported in [IpD93] when state reduction based on symmetry was applied to the verification of a cache coherence protocol. In [CGL92] verification of a pipelined ALU circuit design containing more than 10^{1300} states was reported.

1. Eliminating Redundant Interleavings

An example of state space reduction based on symmetry is available from the mutual exclusion algorithms used in Chapter II. Each program contained two processes P1 and P2. These two processes are symmetric in their actions. Enumerating all possible interleaving generates one state with P1 in its critical section and P2 waiting to enter and another state with P2 is in its critical section while P1 waits. From an analysis perspective these two states are equivalent in the sense that one process is in its critical section and the other is waiting to enter. Which particular case is examined in the verification process is not important, the same results are obtained from either choice. This is an example of eliminating redundant interleavings discussed in [ChHa94] and exploiting symmetry covered in [IpDi93].

2. Folding Related States

Abstraction is another method for reducing the state space by considering the domains of the concurrent system's variables. It may be possible to partition a variable's domain. Instead of using the full range of a variable it may be only necessary to examine situations where the variable is greater or less than some particular value. 'Folding related states' is also referred to as *abstract interpretation*, see [ChHa92], [ChHa94], and [CGL92]

A variation of the mutual exclusion problem can be used to illustrate state folding. Consider a mutual exclusion implementation for N processes that uses a queue to store the process id of all processes waiting to enter its critical section. Let a variable queue_count represent the number of queued processes. To individual processes, the specific process id's contained in the queue and their ordering is generally not significant. What is important is whether the process requesting entry is allowed to enter its critical section (i.e. queue_count is zero). The full domain of the variable queue_count does not need to be modeled. As an approximation, queue_count could be replaced by a boolean variable that would indicate yes or no when a process requests entry into its critical section.

This granularity of abstraction may not be suitable for all analyses. For example, if a particular process execution sequence is required to ensure the correct operation of the overall system (say process P3 must complete its critical section prior to P10 entering its critical section) then an abstraction level that prevents checking the queue's ordering of process ids is unacceptable. An essential detail of the system has been lost.

3. Down-scaling

"One of the most important ways to make verification of large systems possible is down-scaling -- pretending that they are small systems." Dill argues [DDHY92]. The idea of down-scaling is to conduct the verification using a subset of the concurrent system. When a system is scaleable, analysis precision is not lost. Results obtained from work with a subset should reflect problems that exist in the full-scale system.

The concept of down-scaling can be extended further. When a system consists of discrete phases, each phase can be analyzed separately. For this approach to succeed each phase must have a distinct start and end. This idea was used successfully in the verification of SNR conducted for this thesis.

C. COMMUNICATION PROTOCOLS

The parallel actions occurring in protocols makes reasoning about their design difficult. Adding to the inherit complexity of the interactions among the protocol's components are the affects of the transmission media. Data and control errors can be introduced by the channel. Also, there can be a significant time delay from when information is sent until it is received. Specific problems that can be introduced by a network include:

- Data packets delivered out of order.
- Variable round trip delay.
- An intermittent or broken connection.
- Data corruption.

When a verification technique is applied to a protocol, both its concurrent nature and the characteristics of the communication channel should be considered.

The verification process can be simplified if the communication channel is assumed to be perfect (i.e. reliable, packets are always delivered in order, constant delay time, congestion free, data is never corrupted by the channel, etc.). This is a fairly reasonable assumption in the context of fiber optic networks. Invoking this restriction can facilitate the initial verification of a protocol design. However, even under the most ideal conditions network problems can occur. After an initial attempt, verification should be applied to more complex characteristics of a specific target network. It should be kept in mind that if all relevant details are included, complete verification of a protocol may not be feasible.

IV. THE MURPHI VERIFICATION SYSTEM

A. OVERVIEW

The Murphi Verification System is a tool designed to facilitate the verification of finite-state asynchronous concurrent systems. Murphi consist of a **description language** and **compiler**. The Murphi Description Language furnishes a fairly rich set of features for characterizing the behavior and properties of concurrent systems. Two important constructs of the descriptive language are **rule** and **invariant**. Invariants are used to express system properties. Rules portray a portion of the system's overall behavior. Carrying out the action of a rule usually changes one or more system variables, resulting in a transition to another state. The Murphi compiler is used to create an executable program that automatically tests invariants and error conditions while generating all reachable system states.

Four steps are required to use Murphi. First, the concurrent system's specification is translated into the descriptive language recognized by Murphi. Next, the Murphi description is transformed into C++ code using the Murphi compiler. The C++ code produced contains code to generate all allowed state transitions, (i.e., the behavior of the concurrent system) and, code to check for error conditions and the violation of invariants contained in the source description (i.e., the properties of the concurrent system). The C++ code is then complied with a standard C++ language compiler, creating an executable program. The executable program is referred to as the "special purpose verifier". Running the program results in either a verification of the concurrent system or a simulation of the system's execution.

In the next section, a summary of the Murphi Description Language is given. The following section describes the basic operation of the special purpose verifier. In the last section Murphi is applied to the already familiar, mutual exclusion example from Section II.B.4. The Murphi description and the output produced for the example are explained in detail. This demonstration of the tool in the context of a familiar and relatively simple program should provide the necessary background to understand the application of Murphi to SNR. Additional information about Murphi can be found in [DDHY92] and [MeDi93][2]

B. MURPHI DESCRIPTION LANGUAGE CONSTRUCTS

The Murphi description of a typical concurrent system's specification has four parts: a declaration part; a rules' section; a startstate portion; and a collection of invariants. For the most part, the syntax and semantics of the expressions, statements, and declarations used in the description language are similar to those of general purpose programming languages, such as Pascal, C, and Ada. Their employment is usually straightforward, and no explanation is required. However, since **rules, invariants, startstate**, and **rulesets** are not found in general purpose languages, they will be explained in detail.

1. Declaration Part

The declaration part contains definitions for constants, data-types, variables, procedures and functions used in a description. Constants and types are declared first. They are then used in the declarations of global and local variables. Types that can be defined by the user include: simple types -- enumerations and finite integer subranges; and compound types -- arrays and records. Boolean is the only predefined type available.

A potentially powerful feature found in Murphi is a data-type called **Scalarset**. When the concurrent system being investigated exhibit's symmetry with respect to one or more variables, these variables can be declared type scalarset. The use of a scalarset variable allows Murphi to automatically reduce the system's state-space.

[2] The user manual and executable software are available via ftp from Stanford University.

2. Rules

Rules come after the declarations in a Murphi description. They are used to describe the conditions under which transitions are allowed to take place and the action to occur in each global state. Rules have the form:

```
Rule "name"
        expression
==>
Begin
        statement(s)
End;
```

The *expression* is a guard and must evaluate to a Boolean value. The kinds of operators available are quite extensive and include such things as logical implication, and universal and existential quantification. If the expression is true, then the rule's body is executed. The sequence of statements between the keywords **begin** and **end** comprise the body of the rule. A rule's entire body is executed indivisibly. Example of statements available in Murphi are: *Switch* statements; *for* and *while* loop statements; procedure and function calls; assertions; and output statements.

Rules can be grouped in a set by the **Ruleset** construct. It has the form:

```
Ruleset  identifier: range of identifier Do
        set of rules
Endruleset;
```

The variable *identifier* is local, and only effects those rules within the *set of rule*. As the verifier executes, *identifier* takes on each value in its range. A ruleset essentially duplicates the behavior of the individual rules contained in the *set of rules* for each value in *range of identifier*.

3. Startstate

Variable initialization is accomplished using a special rule called **Startstate**. All variables must be given initial values by the startstate rule. The startstate is only executed once, at the beginning of the verification process.

4. Invariant

Invariants are used to specify properties of the concurrent system. They have the form:

> **Invariant** "*name*"
> *expression*

The *expression* must evaluate to type Boolean. Its value will be checked in each state generated during execution of the verifier.

Similar to **Invariant** is a construct called **Liveness**[3]. With **Liveness** a property can be written using a subset of Linear Time Temporal Logic (LTL). The temporal operators, EVENTUALLY, ALWAYS, and UNTIL, are supported.

C. SPECIAL PURPOSE VERIFIER

1. State Generation and Property Checking

The executable program generated from the Murphi description, called the special purpose verifier, (or just verifier) can run in two modes -- verification or simulation. The first rule executed in either mode is the **Startstate** rule. The body of **Startstate** initializes all system variable and defines the first global state available to the verifier. Based on the values of the system variables, the verifier identifies all rules with guards that evaluate to true. The body of each enabled rule is executed. The state generated is checked for various error conditions (i.e., run-time errors, deadlock, violation of user-defined asserts statements, error statements, and invariants). If no error is detected, then these newly generated states are inserted into a queue. After all enabled rules have fired, a state is extracted from the queue to become the new system state. Execution of the verifier then continues from this state. The process of determining all state transition rules satisfied, generating the new states, checking for errors in those states, inserting the states into the queue, and extracting a state from the queue, is repeated. In the verification mode, the

[3] Liveness is supported in Murphi Version 2.7L

user can select either a depth-first or a breadth-first state generation path. During execution, Murphi chooses among the enabled rules arbitrarily to generate the next state. If an error condition occurs, the verifier halts and reports the cause, otherwise its termination depends on the run mode. In the verification mode the verifier runs until all states have been generated. In the simulation mode it continues to execute until terminated by the user.

2. Execution Report

When ran in the verification mode, the verifier displays, every 1000 events, the number of states explored and the amount of time expended. When execution is complete, the errors encountered and the total size of the state space explored are reported. In the simulation mode the verifier normally displays the total number of rules fired every 1000 event. Murphi has various options available for controlling its output. The level of detail in the execution report can be increased, decreased or changed to meet the user's need. Examples of these options include: "make simulation or verification verbose"; "print out rule information"; and "write a violation trace".

D. APPLICATION OF MURPHI TO MUTUAL EXCLUSION

1. The Murphi Description

The verification process begins by translating a concurrent program to a Murphi Description. For this example, the mutual exclusion algorithm discussed in Section II.B.4 has been translated into the descriptive language recognized by Murphi. The description is presented on the next two pages. Note the correspondence between the concurrent program (shown in the box) an its description.

```
-- Murphi Description of MUEX algorithm
-- exhibiting deadlock

--Declarations

Type
  test_var_type: 0..1;
  P1_label_t: Enum{L1_1,  -- non critical section
    L1_2,  -- assign C1 := 0
    L1_3,  --  loop while other process in critical
section
    L1_4,  -- critical section
    L1_5   -- assign C1 := 1 };
  P2_label_t: Enum{L2_1,  -- non critical section
    L2_2,  -- assign C1 := 0
    L2_3,  -- loop while other process in critical section
    L2_4,  -- critical section
    L2_5   -- assign C1 := 1 };
Var
 P1: P1_label_t;
 P2: P2_label_t;
 C1: test_var_type;
 C2: test_var_type;
```

```
Portion of Program

subtype TEST_VAR_TYPE is integer range
0..1;
C1, C2: TEST_VAR := 1;

task body P1 is
begin
   loop

L1.1  Non_Critical_Section_1;
L1.2  C1  :=  0;
L1.3  loop  exit when C2 = 1;   end loop;
L1.4  Critical_Section_1;
L1.5  C1  :=  1;

   end loop;
end P1;
```

```
-- Rules

 Rule "P1 non-critical section"
  P1 = L1_1  ==>
 Begin
  P1 := L1_2;
 End;

 Rule "P1 assign C1 0"
  P1 = L1_2  ==>
 Begin
  C1 := 0;
  P1 := L1_3;
 End;

 Rule "P1 wait"
  P1 = L1_3  ==>
 Begin
  If (C2 = 1)  Then
    P1 := L1_4;
  End; --If
 End;

 Rule "P1 critical section "
  P1 = L1_4  ==>
 Begin
  P1 := L1_5;
 End;

 Rule "P1 assign C 1"
  P1 = L1_5  ==>
 Begin
  C1 := 1;
  P1 := L1_1;
 End;

 Rule "P2 non-critical section"
  P2 = L2_1  ==>
 Begin
  P2 := L2_2;
 End;
```

```
Rule "P2 assign C1 0"
 P2 = L2_2  ==>
Begin
 C2 := 0;
 P2 := L2_3;
End;

Rule "P2 wait"
 P2 = L2_3  ==>
Begin
 If (C1 = 1)  Then
   P2 := L2_4;
 End; --If
End;


Rule " P2 critical section "
 P2 = L2_4  ==>
Begin
 P2 := L2_5;
End;
```

```
Rule "P2 assign C2 1"
 P2 = L2_5  ==>
Begin
 C2 := 1;
 P2 := L2_1;
End;


-- initialization
Startstate
 P1 := L1_1;
 C1 := 1;
 P2 := L2_1;
 C2 := 1;
End;


-- safety property
Invariant "Mutual Exclusion Violated"
 ! (P1 = L1_4 & P2 = L2_4)
```

a. Declarations

Three data-types are declared. The statement 'test_var_type: 0..1;' specifies an integer subrange with domain {0, 1}. The next declaration is an enumeration-type, called 'P1_label_t' with domain {L1_1, L1_2, L1_3, L1_4, L1_5}. 'P2_label_t' is also an enumeration-type with domain {L2_1, L2_2, L2_3, L2_4, L2_5};

Next to appear is the declaration of variables. Four variables are declared. The first variable, 'P1', represents the "program counter" of process P1 and can take on values of type 'P1_label_t'. Similarly, the second variable represents the "program counter" of process P2. The two variables C1 and C2 are of type 'test_var_type' and can be assigned a value of 0 or 1. They serve a binary semaphores.

b. Rules

The first rule in the description, named "P1 non-critical section", is enabled if its guard 'P1 = L1_1' is true. The action of its body is to assigns P1 the value L1_2.

Behavior of the other rules is similar to the first rule. The table below explains the purpose of each rule.

Name	Purpose
P1 non-critical section	When at *L1.1*, advance P1's program counter to *L1.2*
P1 assign C 0	When at *L1.2*, assign C1 the value 0, advance P1's program counter to *L1.2*
P1 wait	When at *L1.3*, check the value of C2, if C2 = 1 then advance P1's program counter to *L1.4*, if C2 = 0 then the program counter remains at *L1.3*
P1 critical section	When at *L1.4*, advance P1's program counter to *L1.5*
P1 assign C 1	When at *L1.5*, assign C1 the value 1, change P1's program counter to *L1.1*
P2 ...	Analogous to the first five, except for process P2

Table 2. Explanation of Rules for Mutual Exclusion Description.

The rules used in this example are very simple. Rules can be much more involved. For example, a rule's guard can consist of a complicated expression. Also, declarations of local variables, constants and types can be inserted between the rule's condition and body.

c. ***Startstate***

In this example four variables must be initialized. The initial value of these variables are as expected for the MUEX program. P1 and P2 start at *L1.2* and *L2.1*, respectively. C1 and C2 are both assigned a value of 1.

d. ***Invariants***

This example has only one invariant. The invariant's name is 'Mutual Exclusion Violated'. The expression is read as : $\neg((P1 = L1\_4) \wedge (P2 = L2\_4))$. It is false (i.e., the invariant is violated) if a state is generated where both P1 and P2 are in their critical sections. After each new state is generated, the invariant is checked. If false, execution of the verifier terminates and a report is displayed.

2. Murphi's Output

Below is the report generated by the special purpose verifier produced from the Murphi description presented in paragraph IV.D.1. Only the first few states and the final states of the report are shown. See Appendix A for a complete listing of this report.

```
Verbose option selected.
The following is the detailed progress.

Firing startstate Startstate 0
Obtained state:
P1:L1_1
P2:L2_1
C1 : 1
C2 : 1

-----------------------------
Unpacking state from queue:
P1:L1_1
P2:L2_1
C1 : 1
C2 : 1

The following next states are obtained:

Firing rule P2 non-critical section
Obtained state:
P1:L1_1
P2:L2_2
C1 : 1
C2 : 1

Firing rule P1 non-critical section
Obtained state:
P1:L1_2
P2:L2_1
C1 : 1
C2 : 1

-----------------------------
Unpacking state from queue:
P1:L1_1
P2:L2_2
C1 : 1
C2 : 1

The following next states are obtained:

Firing rule P2 assign C1 0
Obtained state:
P1:L1_1
P2:L2_3
C1 : 1
C2 : 0

Firing rule P1 non-critical section
Obtained state:
P1:L1_2
P2:L2_2
C1 : 1
C2 : 1

-----------------------------
Unpacking state from queue:
P1:L1_2
P2:L2_1
C1 : 1
C2 : 1

The following next states are obtained:
```

... skipping to the last few transitions of the trace report ...

```
-----------------------------
Unpacking state from queue:
P1:L1_3
P2:L2_3
C1 : 0
C2 : 0

The following next states are obtained:

Firing rule P2 wait
Obtained state:
P1:L1_3
P2:L2_3
C1 : 0
C2 : 0

Firing rule P1 wait
Obtained state:
P1:L1_3
P2:L2_3
C1 : 0
C2 : 0
==========================================================================

Result:
        Deadlocked state found.

State Space Explored:
        17 states, 26 rules fired in 0.40s.

Rules Information:
        Fired 1 times     - Rule "P2 assign C2 1"
        Fired 2 times     - Rule " critical section "
        Fired 3 times     - Rule "P2 wait"
        Fired 3 times     - Rule "P2 assign C1 0"
        Fired 4 times     - Rule "2P non-critical section"
        Fired 0 times     - Rule "P assign C 1"
        Fired 1 times     - Rule " critical section "
        Fired 3 times     - Rule "P1 wait"
        Fired 4 times     - Rule "P1 assign C1 0"
        Fired 5 times     - Rule "P1 non-critical section"
```

The first state of the execution path -- (L1.1, 1, L2.1, 1) -- is that defined by the startstate. In this state, two rules are enabled: ‘P1 non-critical section’ and ‘P2 non-critical section’. The body of each of these rules engender separate transitions and produce distinct states as shown below.

Name of rule enabled :	P1 non-critical section	P2 non-critical section
State produced:	(L1.**2**, 1, L2.1, 1)	(L1.1, 1, L2.**2**, 1)

Since no error or violation of the invariant occurred, both of these states are placed on a queue. The state produced from rule 'P2 non-critical section' is extracted first. In this state, the guards of two rules, 'P2 assign C1 0' and 'P1 non-critical section', are true. The states produced by these rules are checked for errors and placed in the queue. The queue now contains three states {(L1.2, 1, L2.1, 1), (L1.1, 1, L2.3, 0), (L1.2, 1, L2.2, 1)}. (Note, even though the execution of a rule is repeated -- 'P1 non-critical section' -- a different state is obtained since in this interleaving, rule 'P2 assign C1 0' has already fired.) The verifier is using a breadth first search strategy, so state (L1.2, 1, L2.1, 1) is chosen, and the verification continues. After ten more states are reached, the state (L1.3, 0, L2.3, 0) is the next state removed from the queue. Two rules are enabled in this state, 'P2 wait' and 'P1 wait'. Execution of the body of either rule produces the state (L1.3, 0, L2.3, 0). However this state is the same as the previous state -- deadlock. Since an error condition has been detected, the verification process stops and a report is displayed. The report shows all states examined, rules fired, and the error detected. The sequence of states leading to the deadlocked condition can be determined and analyzed.

A report tracing the path to an error can be helpful for identifying flaws in the design of a concurrent system. Based on insight gained from an analysis of the verifier's output, it may be possible to modify the concurrent system and prevent occurrence of the execution interleaving that results in the undesirable state.

V. THE SNR TRANSPORT PROTOCOL

A. INTRODUCTION

At the top level, the transport protocol SNR is a set of rules controlling the exchange of data between a **transmitter** and **receiver** connected by a **network**. The transmitter and receiver run in parallel. They cooperate to transfer data from a sending host (interfaced with the transmitter), and the receiving host (linked to the receiver). The transmitter and receiver use **packets** to exchange data and control information.

The data transfer process consists of four basic steps. The transmitter is given data by its host to send. The transmitter encapsulates the data into packets and inserts them into the network for transmission. After the propagation delay intrinsic to the channel, the data packets arrive at the receiver and are extracted from the network. The receiver processes the packets and then delivers the data to its host. This process would be relatively straight forward if not for finite receiver resources and problems[4] introduced by the network. The constraints of the receiver are: 1) an upper bound on the rate at which it can process data packets, and 2) a limit of the size of its buffer. (A buffer is required to temporarily hold and reorder packets prior to delivery to the receiver's host.)

The actions of the transmitter and receiver must be coordinated to reliably transfer data over a network. Information is passed between these two entities to achieve the coordination required to carry out the five functions provided by SNR. These functions are:

- Connection Management -- establishing the connection, detection of an unintended connection termination, and connection termination after completing the data transfer.
- Flow Control -- restricting the number of packets in transit from the transmitter to prevent buffer overflow in the receiver.

[4] Problems that can be introduced by a network include, data corruption, out of order data packets, lost data packets, variable round trip delay, broken connection, etc.

- Error Control -- detecting and recovering from lost packets or corrupted data. SNR employs a modified *selective repeat* error recovery method.
- Ordered Delivery -- delivering data packets to the receiver's host in the sequence sent by the transmitter.
- Multiplexing/Demultiplexing -- establishing and communicating on more than one connection at a time. (Multiplexing/Demultiplexing will not be covered in this thesis.)

The description of SNR's organization and operation will be introduced in steps. First a block diagram of SNR is presented. Second, a brief overview of the protocol's operation is given. This is followed by a description and explanation of connection parameters, packet formats, variables, and data structures used in SNR. Next, state transition diagrams of the machines internal to the transmitter and receiver are presented. Finally, a detailed example of a portion of a typical data transfer session is given to illustrate the actions of these machines.

The following concepts are useful to keep in mind when reading the following sections:

- The transmitter attempts to send as many data packets as possible without overflowing the receiver's buffer. When the transmitter believes the receiver's buffer is full, the transmitter must halt transmission of data packets and wait until a receiver control packet arrives acknowledging blocks previously sent.
- The transmitter must retain a copy of data already sent (in case retransmission is required) until that data has been acknowledged by the receiver.
- The state of the receiver, as known by the transmitter, is never current. Any state information sent by the receiver takes a finite amount of time before it gets to the transmitter.

B. DESIGN FEATURES

Most transport protocols in use today fail to deliver the performance expected with networks utilizing advanced components such as fiber optics. Existing protocols, for the most part, were conceived and implemented prior to the development of technologies used in modern networks. To overcome the deficiencies present in older, less reliable, and slower networks, current protocols employ complex control procedures and thus suffer from high processing overhead. The large processing demands placed on a system running an inefficient protocol, reduces its ability to transfer data to and from the network. This creates a mismatch between the communication channel's capacity for sustained high transmission rates and the system's slower throughput. The transport protocol SNR has been proposed to address this problem. It is specifically designed to take advantage of the extended bandwidth, high speed switching, and lower error rate of modern networks.

The design goal of SNR is to increase its overall performance while still coping with problems that can be encountered even in modern networks. Two primary innovations in SNR's design aim to achieve this goal. They are:

- frequent and periodic exchange of complete state information between the transmitter and receiver, and
- flow and error control based on packets grouped in blocks vice individual packets.

The concepts of **periodic state exchange** and **blocking** are intended to simplify the protocol's overall design, diminish its processing demands, and permit an implementation based on parallel processing. Parallel processing coupled with lower processing overhead should significantly increase the throughput of the system running the protocol. The expected result is a faster transport protocol even in the presence of transmission errors.

1. Periodic State Exchange

SNR exchanges complete state information of the transmitter and receiver frequently and periodically apart from the occurrence of significant events. Most other protocols pass the status of the transmitter or receiver only after detecting an error such as a lost data packet. The error detection procedures typically involve explicit round-trip delay timers, large data structures and complex packet acknowledgment schemes. Decoupling state exchange from specific events and frequently passing complete state information, reduces the protocol's processing requirements for two reasons.

First, the loss of a control packet (a packet containing state information not data) has no significant impact in SNR. A new one, with the same or more current information, will be along shortly. Other protocols must have some means, usually complicated, for dealing with lost state packets since the information in each of these is accumulative. With most protocols the information in the most recent control packet augments a history of state information. In SNR, the information of individual control packets is complete and can be processed independently of previously transmitted control packets.

Second, frequent and periodic transmission of control packets can be used to implement implicit timers. SNR uses simple counters to achieve the functionality of clock based timers. The control packets are transmitted at a frequency linked to the reception rate of data packets. Each time a control packet is received, a counter is incremented. Thus the interval between control packets and the rate at which a counter is altered roughly corresponds to the current round trip delay of the network. With this approach, SNR can automatically adjust to varying network conditions. SNR uses these "implicit timers" for its retransmission and broken connection timers. The elimination of explicit, clock-based, round-trip delay timers, and their associated problems[5] potentially provides the greatest gain from using periodic state exchange. See [Zha86] for a detailed discussion of timer problems in network protocols.

It may appear that passing state information in a fashion as in SNR might reduce the throughput of the system. After all, this approach places extra packets into the communication channel. However, it must be kept in mind that the transmission rate of the channel is not limiting. Since a high speed network is normally running below capacity, a protocol design that speeds up the transmitter and receiver, even though additional packets are generated, should increase the achievable overall data transfer rate.

[5] The problem with explicit timer is: to what value should it be set? Too small, and unnecessary retransmissions occur. Too large and the protocol responds too slowly to a lost packet. Any static timer setting strategy will be unable to respond to changing network conditions. Proposed schemes to dynamically modify the value so far have failed to provide an adequate solution.

2. Blocks of Packets

To take advantage of a fiber optic network's low error rate, SNR implements a block-based flow control and error control scheme. Rather then acknowledging and retransmitting individual data packets, groups of packets are managed. This approach has two effects on the data transfer process. First, the size of tables and the complexity of the procedures used by the protocol to track the status of data packets are reduced. Second, the number of packets sent during a session may increase because blocks (all packets in the block not just the single lost or damaged packet) are retransmitted when data is lost or corrupted. The first has a positive impact on the protocol's performance while the second tends to reduce its throughput. In networks with low error rates, the retransmission of a full block should not occur very often and therefore unnecessary packets are sent very infrequently. The processing speedup is expected to outweigh the higher packet count, resulting in better overall efficiency compared to a non packet-blocking protocol.

3. Operating Modes

SNR's design allows the level of service provided by the protocol to be controlled. Three operating modes are available in SNR. In **Mode 0**, SNR runs with flow and error control omitted. In **Mode 1**, flow control is provided but not error control. Both error and flow control function in **Mode 2**.

The reliability of the network and the type of data being transferred influences mode selection. Mode 0 is used when a fast data transfer rate without concern for errors is the principal objective. Mode 1 is best suited for real time applications. The preferred choice for transferring large files over a network likely to introduce errors is Mode 2. According to [NRS90] the efficiency of SNR is optimized when large packets are used in Mode 2, and small sized packets with Modes 0 and 1.

C. THE SNR ARCHITECTURE

The formal specification for SNR, provided in [NRS90], is based on a finite-state machine model. The protocol is specified by seven machines, three machines for the transmitter (T1, T2, T3) and four for the receiver (R1, R2, R3, R4). The machines internal to the transmitter and the receiver are intended to run in parallel without explicit synchronization. The machines cooperate to pass data from the transmitter to the receiver. Their actions are coordinated by means of shared variables and message passing. A block diagram of SNR is displayed in Figure 2 and a table summarizing the purpose of each machine is presented in Table 3. The arrows in the diagram represent information flow across the network accomplished by passing messages.

D. OVERVIEW OF SNR'S OPERATION

Below is a sequence of actions performed during a data transfer session under SNR. Most of the details have been omitted. See the last section of this chapter for an example with actions at the state transition level of the internal machines.

1. The transmitter's host signals T2 that it has a message to send.
2. T2 and R2 negotiate the parameters for the session and establish the connection.
3. T1 transmits blocks of new data packets until the preset limit on the capacity of the receiver's buffer is reached or retransmission of a block is required.
4. R1 processes the incoming data packets and updates the receiver's tables used for tracking the reception status of packets and blocks. These two tables indicate the need for data retransmission.
5. At the appropriate interval, R3 sends receiver state information to T2. This information is used to update the status of the receiver's buffer (as known by the transmitter) and acknowledge blocks of data.
6. T3 periodically sends transmitter state information to R2. There are a set number of blocks transmitted between each control packet.
7. R1 reorders the data packets as appropriate.
8. The processed packets are passed to the host by R4.
9. Control packets continue to be exchanged and blocks of data packets sent until the entire message has been acknowledged by the receiver.

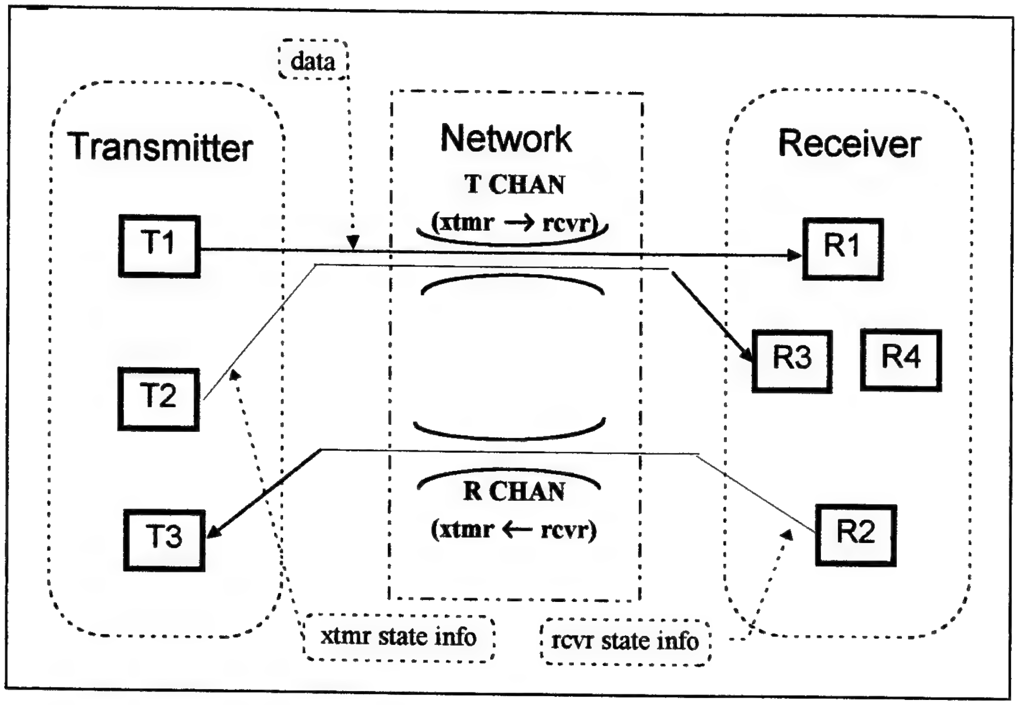

Figure 2. Block Diagram of SNR.

Machine	Purpose
T1	Transmits/retransmits data packets.
T2	Manages the connection and flow control for the transmitter.
T3	Sends the transmitter's state information to the receiver.
R1	Processes incoming data packets.
R2	Manages the connection and flow control for the receiver.
R3	Sends the receiver's state information to the transmitter
R4	Passes processed data packets to the host.

Table 3. Purpose of Each Machine in SNR.

10. T1 will retransmit a block of data packets if the receiver fails to acknowledge a packet from that particular block prior to its retransmission counter expiring.

11. T1 temporally halts transmission of data packets if its information indicates the receiver's buffer will be full when all of the data packets it has sent arrive at the receiver. T1 resumes sending data packets when state information from the receiver indicates buffer space is once again available.

12. T3 will terminate the connection if it has not received a receiver control packet within the required time limit.

E. COMMUNICATION PARAMETERS AND STRUCTURES

1. Connection Parameters

Parameters particular to each connection are determined during the connection establishment phase. They include: number of bits per packet; number of packets per block; buffer size; round trip delay (RTD); and bandwidth. Their values are then used for calculating other connection parameters and initial values for protocol variables. Discussed below are important connection parameters calculated by SNR.

a. L -- Largest Allowed Number of Outstanding Blocks

The value of L is chosen be slightly larger than

$$\left(\frac{RTD \times bandwith}{bits\ per\ block}\right).$$

For example, assuming:

RTD = 20 msec	bandwidth = 1 Gbit/sec
1000 bits per data packet	8 packets per block,

for the connection gives the result

$$\left(\frac{\left(20\times10^{-3}\,\text{sec}\right)\times\left(1\times10^{9}\,bits\,per\,\text{sec}\right)}{\left(1000\ bits\,per\,packet\right)\times\left(8\ packets\,per\,block\right)}\right)=2500\ blocks.$$

Based on these values, L must be greater than 2500 blocks.

b. T_{in} -- Periodic Time Interval

Control packets are transmitter at interval

$$T_{in} = \mathbf{max}\left(\frac{RTD}{kou}, IPT\right).$$

The constant *kou* is typically a power of 2 such as 32, and IPT is the average time between the transmission of two data packets. The value of T_{in} changes when the connection becomes inactive. If a data packet has not been sent within the period T_{in}, the value of T_{in} is increased by a factor of 2. While the connection remains inactive, T_{in} continues to increase by a factor of 2. However, it never exceeds the maximum of the either $\frac{RTD}{m}$ or IPT, where *m* is another constant such as 8. The value of T_{in} immediately changes back to $\mathbf{max}\left(\frac{RTD}{kou}, IPT\right)$ when data packet transmission resumes.

For example, using RTD = 20 msec, *kou* = 32 and IPT = 0.05 msec gives,

$$T_{in} = \mathbf{max}\left(\frac{20\times10^{-3}\text{ sec}}{32}, 0.05\times10^{-3}\text{ sec}\right) = 0.625\text{ msec}.$$

If the connection is inactive T_{in} increase to 1.25 msec, then to 2.5 msec.

2. Packets

The formats of the packets used by SNR for transferring data and exchanging state information over the network are shown below. Following the packet formats are descriptions of the fields comprising the packets (Table 4).

Packet Type	Format
Data Packet	LCI | Type=2 | Seq# | Data | Error Check
Transmitter Control Packet	LCI | Type=1 | Seq# | k | UW$_t$ | No. of blocks queued | Error Check
Receiver Control Packet	LCI | Type=0 | Seq# | k | LW$_r$ | Buffer_available | LOB | Error Check

FIELD NAME	PURPOSE
LCI	Logical Connection Identifier, indicates with which connection the packet is associated. Only significant when multiple connections are established.
Seq#	The packet's sequence number.
Data	Contains the data being transferred. The number of bits used for this field is negotiated during connection establishment.
Type	Receiver control packet - **0**, Transmitter control packet - **1**, Data packet - **2**.
k	The interval between sending two sequential state control packets in units of T_{in}.
UW_t (Upper Window Transmitter)	Sequence number of the highest block transmitted but that may not have been acknowledged. (UW_r is analogous.)
LW_r (Lower Window Receiver)	Every block with a sequence number less than this number has been acknowledged. (LW_t is analogous.)
No. of blocks queued	The number of blocks that have not yet been transmitted.
Buffer_available	The space remaining in the receiver's buffer (in blocks).
LOB	Table of Outstanding Blocks - A bit map maintained by the receiver indicating the outstanding blocks in its window.
Error Check	Error detection code.

Table 4. Fields of SNR's Packets.

3. Shared Variables

Presented below are the primary variables of the transmitter and receiver used in the implementation discussed in [NRS90]. These variables are local to either the transmitter or receiver and used for coordinating the actions of their internal machines.

VARIABLE	PURPOSE
Start_signal	Sent by T2 and R2 to indicate the connection has been established.
busy	In the transmitter it indicates whether a data packet has been sent recently. In the receiver it indicates whether a date packet has arrived recently.
clock_tick	Periodic event occurring at interval T_{in}.
scount	Counter employed to implement a timer used to detect a broken connection or a failed transmitter or receiver.
count	Counter used to implement a timer that marks the interval between sending control packets.
k	The interval, in units of T_{in} between sending two sequential state control packets.
LUP	A table used by the transmitter to maintain the acknowledgment status of transmitted blocks. It has three fields for each element, [Seq#, count, ack].
buffer_available	The amount of space available in the receiver's buffer, as known by the transmitter. This variable is updated with the information in the buffer_available field contained in receiver control packet.
NOU	The number of blocks sent by the transmitter but not yet acknowledged by the receiver. *NOU* must always be less than *L*.
AREC[.]	A table used by the receiver to maintain the status of received **blocks**. AREC[i] is set to 1 when all packets in block 'i' received error free.
RECEIVE[.]	A table used to maintain the status of received **packets**. RECEIVE[j] is set to 1 when packets 'j' received error free.

Table 5. Variables and Data Structures.

Additional details of the variables required for the operation of the connection establishment phase and flow control will be provided, as appropriate, in Chapter VI and Chapter VII respectively.

F. THE SNR MACHINES

A state transition diagram of each machine used in SNR is given below along with an explanation of its transition. These diagrams mimic the FSM's given in [NRS90] . They are provided as a means of illustrating the concurrent action of the various entities comprising SNR and are not intended to serve as a specification.

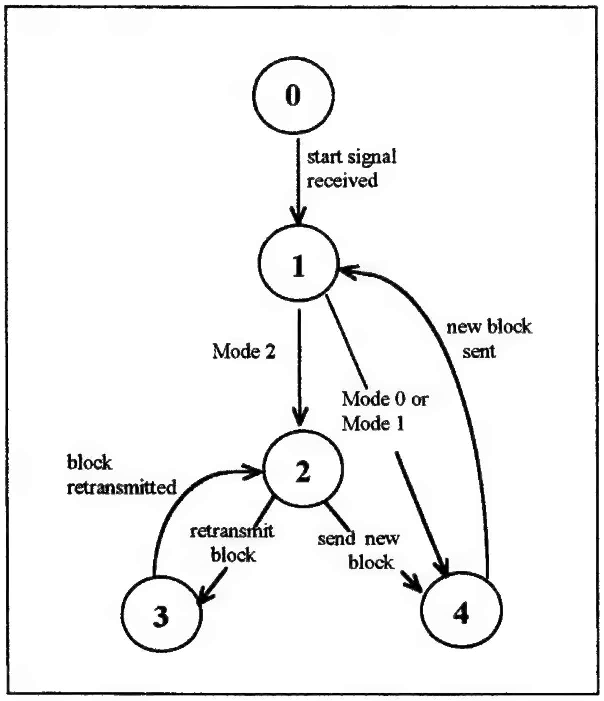

Figure 3 Machine T1 State Diagram

TRANSITIONS	EXPLANATION
0 → 1	Occurs after start_signal received from T2.
1 → 2	Occurs if Mode 2 (flow control and error control) is being used.
1 → 4	Occurs if Mode 0 (no flow or error control) is being used, or if Mode 1 is being used and information at the transmitter indicates there is space in the receiver's buffer for a block of data packets.
2 → 3	Occurs if the retransmission counter for an outstanding block reaches zero.
2 → 4	Occurs if there are no outstanding blocks to retransmit, and the receiver's buffer has space for another block even after all blocks in transit have arrived.
3 → 2	Occurs after an outstanding block has been retransmitted and the variable *busy* has been set to true.
4 → 1	Occurs after the transmitter has: sent a new block; updated the table of outstanding blocks (*LUP*); and signals T3 that a block of data has been sent (*busy* set to true).

Table 6 Transitions for Machine T1.

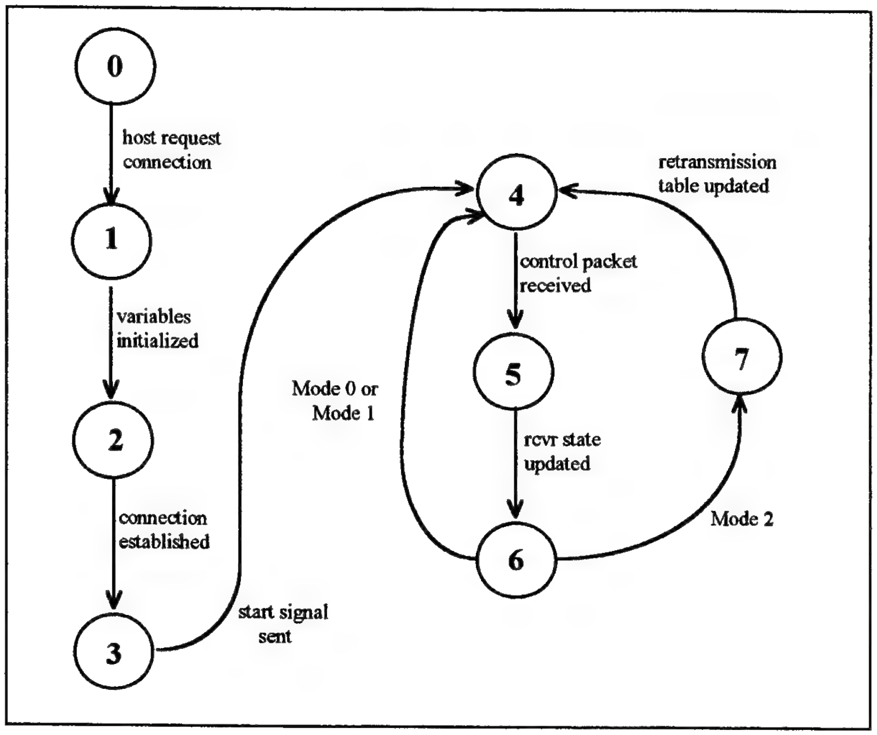

Figure 4 Machine T2 State Diagram.

TRANSITION	EXPLANATION
$0 \rightarrow 1$	Occurs after a connection request is received by T2 from the transmitter's host.
$1 \rightarrow 2$	Occurs after *scount*, UW_t, LW_t, and *LUP* are initialized.
$2 \rightarrow 3$	Occurs after the connection is established with the receiver.
$3 \rightarrow 4$	Occurs after start_signal is sent to T1 and T3.
$4 \rightarrow 5$	Occurs if a control packet is received from the receiver.
$5 \rightarrow 6$	Occurs after updating the receiver's state information maintained at the transmitter and the disconnect counter (*scount*) has been reset.
$6 \rightarrow 4$	Occurs if Mode 0 or Mode 1 are being used.
$6 \rightarrow 7$	Occurs if Mode 2 is being used.
$7 \rightarrow 4$	Occurs after updating the block retransmission table *LUP*.

Table 7. Transitions for Machine T2.

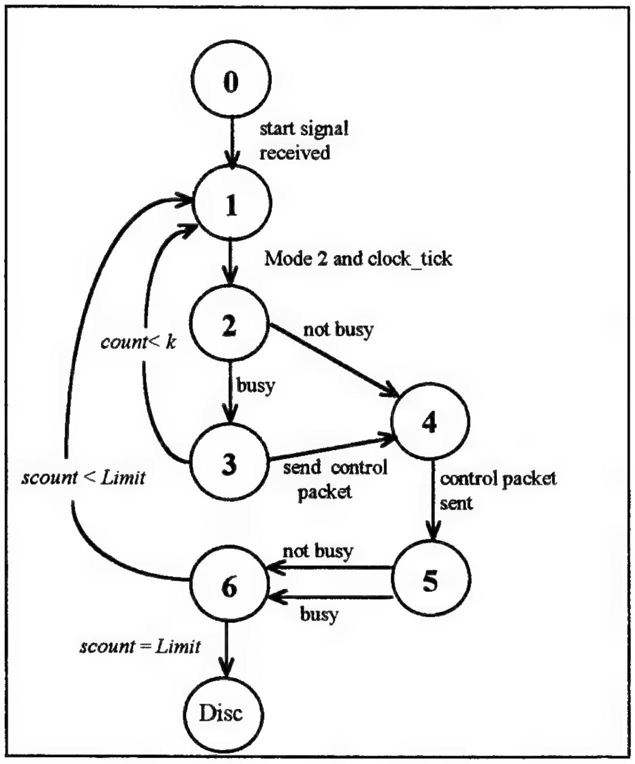

Figure 5. Machine T3 State Diagram.

TRANSITION	EXPLANATION
0 → 1	Occurs after receiving start_signal and variables *k* and *count* are initialized.
1 → 2	Occurs if Mode 2 is being used and the periodic event, *clock_tick*, is detected. Additionally, the shared variable *scount* is incremented.
2 → 3	Occurs if the transmitter has sent data since the last occurrence of *clock_tick*. Additionally, the shared variable *count* is incremented.
2 → 4	Occurs if the transmitter has not sent any data since the last *clock_tick*.
3 → 4	Occurs if *count* = *k*, indicating the transmitter's current state is required to be sent.
3 → 1	Occurs if *count* < *k*.
4 → 5	Occurs after the transmitter's state has been sent and *count* reset to zero.
5 → 6 not busy	Occurs if the transmitter has not recently sent data (*busy* = false). Additionally, *k* is increased to lengthen the interval between control packet transmissions.
5 → 6 busy	Occurs if the transmitter has sent data recently (*busy* = true).
6 → 1	Occurs if *scount* < *Limit* (the disconnect "timer" has not expired) and after *busy* set to false.
6 → *Disc*	Occurs if *scount* = *Limit* (a receiver control packet has not been received in the expected interval and *scount* reached the predetermined value

Table 8. Transitions for Machine T3.

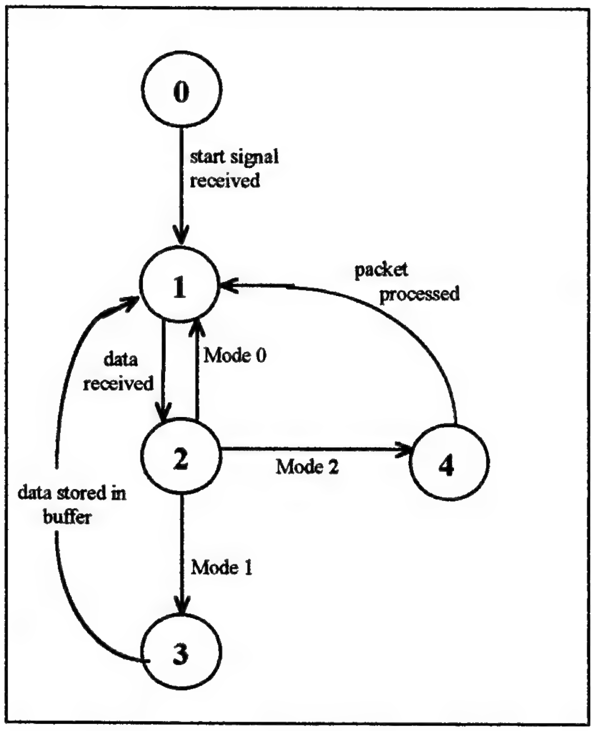

Figure 6. Machine R1 State Diagram.

TRANSITIONS	EXPLANATION
$0 \rightarrow 1$	Occurs after start_signal received from R2.
$1 \rightarrow 2$	Occurs if a data packet is received from the transmitter.
$2 \rightarrow 1$	Occurs if operating in Mode 0. The packet is delivered to the host without any processing in Mode 0.
$2 \rightarrow 3$	Occurs if Mode 1 is being used.
$2 \rightarrow 4$	Occurs if Mode 2 is being used.
$3 \rightarrow 1$	Occurs after data is stored in the receiver's buffer.
$4 \rightarrow 1$[6]	Occurs after the receiver has processed the data packet and updating the two tables *RECEIVE* and *AREC*.

Table 9. Transitions for Machine R1.

[6] The specification for R1 in [NRS90] shows this as transition $4 \rightarrow 2$. However this must be an error because after returning to state 2 from 4, the machine would be stuck in state 2 since Mode = 2, not 1 or 0.

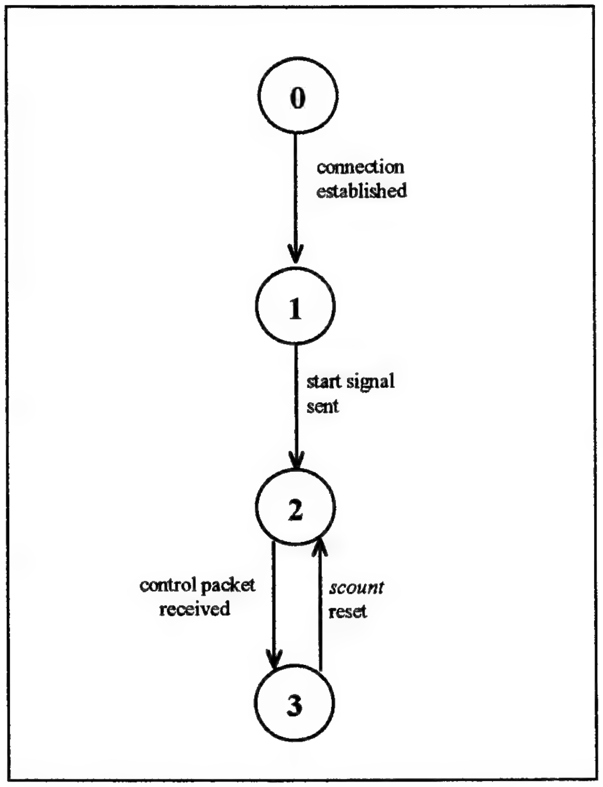

Figure 7. Machine R2 State Diagram.

TRANSITIONS	EXPLANATION
0 → 1	Occurs after the connection has been established with the transmitter.
1 → 2	Occurs after start_signal sent to R1, R3, and R4.
2 → 3	Occurs if a control packet is received from the transmitter.
3 → 2	Occurs after the variable *scount* is reset to zero.

Table 10. Transitions for Machine R2.

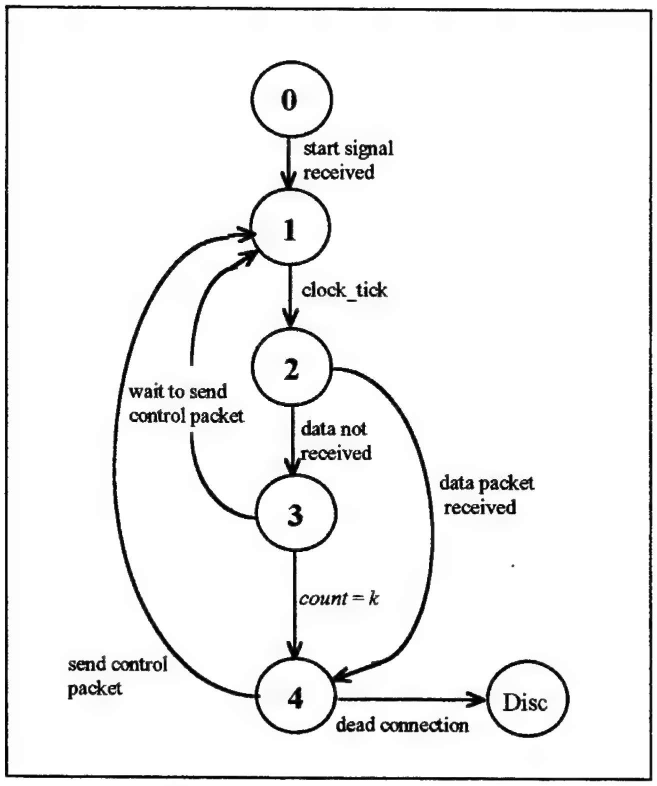

Figure 8. Machine R3 State Diagram.

TRANSITIONS	EXPLANATION
0 → 1	Occurs after start_signal received from R2 and variables initialized (*busy* to false, *k* to 1, and *count* to 0).
1 → 2	Occurs if event *clock_tick* detected and after *scount* incremented.
2 → 3	Occurs if a new data packet has **not** been received and after *count* incremented.
2 → 4	Occurs if a new data packet received and after *count* incremented.
3 → 1	Occurs if it is not yet time to send a control packet (*count* < *k*).
3 → 4	Occurs if *count* = *k* and after *k* has been modified to reduce the transmission rate of receiver control packets.[7]
4 → 1	Occurs after a control packet is sent, and after *busy* and *count* are reset.
4 → *Disc*	Occurs if the receiver has **not** received a control packet from the transmitter in the expected interval (*scount* reached predetermined value).

Table 11. Transitions for Machine R3.

[7] The CFSM diagram in [NRS90] incorrectly indicated that k is modified during the transition from state 4 to state 1 and that a control packet is sent during the transition from state 3 to state 4.

Omitted from [NRS90] are the details of machine R4 and the connection establishment phase. It assumes the connection will be established based on the three-way handshake technique. The specification in [McAr92] includes the details of the connection establishment phase and R4. Additionally it adds another machine to the transmitter, T4, to serve as an interface to the transmitter's host. The specification presented in [McAr92] is refined in [Tipi93] and [LuTi94].

G. THE OPERATION OF SNR'S MACHINES

A fragment of a data transfer session is used to illustrate interactions of SNR's machines internal to the transmitter and the receiver. Only the most basic actions will be shown. It is assumed that no errors are caused by the network during the exchange. For this example assume the transmitter's host (called **source**) has a large file to send to the receiver's host (called **destination**), and data transfer phase Mode 2 will be used. The value of variables will be given only when significant. The state of each machine will be given after the occurrence of major events that modify its state. Additional details relating to an event are provided following the table as appropriate (event numbers marked with an asterisk). Two or more events written in the same table row, indicate that in this particular example, the actions are concurrent.

EVENT		State of Transmitter's Machines after event.			State of Receiver's Machines after event.		
No.	Description	T1	T2	T3	R1	R2	R3
1	Transmitter and receiver idle, no connection.	0	0	0	0	0	0
2	Source signals the transmitter that it has a message to send.		1				
3	Variables initialized.		2				
4*	The parameters of the connection are determined. T2 and R2 establish the connection.		3			1	
5	Start_signal sent by T2 to T1 and T3. Start_signal sent by R2 to R1 and R3.		4			2	
6	Start_signal received at T1 and T2 and at R1 and R3.	1		1	1		1
7	Mode 2 being used for connection.	2					
8	Check of retransmission table indicates there are no blocks to retransmit and information indicates space available in the receiver's buffer.	4					

EVENT		State of Transmitter's Machines after event.			State of Receiver's Machines after event.		
No.	Description	T1	T2	T3	R1	R2	R3
9	T1 transmits a block of data packets, updates the retransmission table, and set *busy* to true.	1					
10	Mode 2 being used for connection.	2					
11	Check of retransmission table indicates there are no blocks to retransmit and the information at the transmitter indicates there is space for another block in the receiver's buffer.	4					
12*	T1 transmits block of data packets, update the retransmission table, and set *busy* to true.	1					
13	*Clock_tick* detected at R3 (0.625 msec since event 6).						2
14	*Clock_tick* detected at T3.			2			
15	*busy* is true in the transmitter, so state 3 in T3 is bypassed. *busy* is false in the receiver, so *count* incremented by R3.			4			3
16	A control packet is sent by T3.			5			
17	T1 is sending data so *k* is not increased by T3.			6			
18	*count* = *k* and since no data has been received at R1, *k* is increased.						4
19	A control packet is sent by R3.						1
20	The disconnect timer has not expired (scount < Limit), so T3 sets busy to false.			1			
21	T1 continues to send data packets, T3 continues to send control packets every 0.625 msec. R3 increased *k* (at event 18), so receiver control packets are sent less frequently until a data packet is received.	1, 2, 4, 1, …		1, 2, 4, 5, 6, 1, …			1, 2, 3, …, 4, 1, …
22	The first data packet arrives at the receiver (approximately 10 msec since event 9). *busy* is set to true.				2		
23	Mode 2 being used for connection.				4		
24	The data packet is processed and *RECEIVE*[1] set to 1.				1		
25	R1 continue to receive and process data packets.				1, 2, 4, 1 …		
26	A transmitter control packet arrives.					3	
27	R2 sets *scount* to 0 and waits for the arrival of another control packet.					2	
28	R3 sends a control packet acknowledging some of the data packets processed by R1.						1
29*	At approximately 20 msec since event 9, data transmission stops while T1 waits for an acknowledgment from the receiver.	2					

EVENT		State of Transmitter's Machines after event.			State of Receiver's Machines after event.		
No.	Description	T1	T2	T3	R1	R2	R3
30	Control packet from the receiver containing acknowledgments is received at the transmitter.		5				
31	Control packet processed by T2.		6				
32	Mode 2 is being used.		7				
33	T2 updates *LUP* and waits for another receiver control packet.		4				
34	Space for a block exist in the receiver's buffer.	4					
35*	Transmission of data packets resumed by T1.	1					

Table 12. Data Transfer Example.

Details of significant events marked with an asterisk:

4* Connection Parameters for this example are:
RTD = 20 msec bandwidth = 1 Gbit/sec 1000 bits per data
packet 8 packets per block L > 2500 blocks $T_{in} = 0.625$
msec

12* Events 10, 11, and 12 are a repeat of events 7, 8 and 9. These actions continue to repeat until retransmission of a block is required, the transmitter has filled the receiver's buffer and must wait until space is again available, or the entire message has been sent.

29* The transmitter has sent 2500 blocks and an acknowledgment has not been received. *buffer_available* was set initially to 2500 block and has not yet changed. The number of outstanding blocks, *NOU,* now equals 2500.
The condition (*buffer_available* - *NOU*) > 0 is no longer true, so T1 must wait in state 2 until control packets from the receiver reflect available buffer space or acknowledge some of the transmitted blocks.

35* The protocol will continue to operate, executing actions similar to those described in the table above, until the entire message is transferred.
The operation of the protocol is significantly more complex then presented above.
Only one of many possible execution interleavings is given and many of the finer details are omitted. The example above demonstrates the difficulty of attempting to investigate the correctness of SNR manually. The next two chapters cover the verification SNR's connection establishment phase and data transfer phase, with the assistance of Murphi.

VI. VERIFICATION -- CONNECTION ESTABLISHMENT PHASE OF SNR

A. INTRODUCTION

This chapter describes the verification of SNR's connection establishment phase. The Murphi Verification System is used to determine if properties attributed to the connection establishment phase remain true in all reachable states. The reader may wonder why formal verification of the connection establishment phase is addressed, considering the designers of SNR omitted its details in a detailed description of the protocol in [NRS90] . Why not just assume the connection establishment phase functions as required, skip its verification, and jump into the analysis of the more interesting data transfer phase? There are four reasons for proceeding with its verification:

1. The function of the connection establishment phase is to prepare the transmitter and receiver for a data transfer session. It is during this phase that connection parameters are negotiated and variables used by SNR are initialized. After it is complete, the protocol should be ready to commence the data transfer phase. The correct operation of the connection establishment phase is necessary for the protocol to function as intended. Therefore verification of this phase is a natural start to verifying SNR.

2. The complexity of the phase is appropriate for the initial application of Murphi to the SNR protocol. Starting out with a simple phase of SNR provides an opportunity to gain insight on the workings of the protocol and a better understanding of how best to employ Murphi for protocol verification. The work can serve as a "stepping stone" for applying Murphi to the more complicated data transfer phase of SNR.

3. A detailed analysis of this phase had already been attempted using the *system state analysis* method [LuTi94]. The system state analysis approach encountered difficulties and was unable to provide a complete analysis of SNR. Problems with the method arose because the role of local variables, such as counters, were ignored. A modification to the techniques was employed in [LuTi94] to overcome this problem and a fairly complete analysis resulted. It is important to determine early whether Murphi will encounter similar difficulties.

4. A comparison of the results obtained with Murphi and the system state analysis method can serve as a "validation" of the mechanical verification approach.

The remainder of this chapter covers five topics. First, a complete and detailed specification of the connection establishment phase is given. Next, the operation of the connection establishment phase is explained. This is followed by a discussion of the significant properties to be verified. The Murphi description of SNR's connection establishment phase is then presented. Finally, the verification results are discussed.

B. SPECIFICATION

The formal specification of the connection establishment phase is provided below. This specification is based on the systems of communicating machine (SCM) model discussed in [McAr92] and the specification of SNR given in [Tipi93]. The SCM model uses a combination of finite state machines with their associated Predicate Action Tables (PAT) to characterize the behavior of a concurrent system. The finite state machines denote the states of individual machines comprising the system, and the allowable state transitions. The PAT describes the enabling predicates and actions for every transition in the system.

Only three of SNR's machines are involved in connection establishment. Two machines from the transmitter participate. Machine T4[8] interfaces with the transmitter's host (Figure 9 and Table 16). T2 is the machine responsible for establishing the connection over the network with the receiver (Figure 10 and Table 17). In the receiver, only one machine, R2, is concerned with this phase (Figure 11 and Table 18). R2 cooperates with T2 to set up the connection.

In this specification, two shared variables, **T_CHAN** and **R_CHAN** are used to represent the network connecting the transmitter and receiver. **T_CHAN** is used for passing data and state information from the transmitter to the receiver. **R_CHAN** is used to pass the receiver's state information to the transmitter. They are both based on a FIFO

[8] The specification given in [McAr92] added this machine.

data structure. **T_CHAN**(front) and **R_CHAN**(front) refer to the head elements of the queues. Additionally, a representation completely faithful to the characteristic of a network would prevent packet delivery prior to a specific time delay. Network propagation delay is ignored in this implementation.

Messages and variables used in this phase of the protocol are described in Table 13 and Table 14, respectively. Non-trivial processing required by an action or predicate, associated with a transition, is performed using a pseudo procedure call. Procedures required for the connection establishment phase are explained in Table 15. Procedure names are in bold type in the PAT's.

MESSAGES		
Name	Flow (From → To)	Purpose
Conn_req	T2 → R2	Connection request, contains the connection parameters desired by the transmitter.
Conn_ack	R2 → T2	Connection acknowledgment, contains the connection parameters the receiver is capable of supporting.
Conn_conf	T2 → R2	Connection confirmation, indicates that the response send by receiver is acceptable to the transmitter.
T_state	T2 → R2	Control packet, contains transmitter's state information.
Data	T2 → R2	Data packet, contains data for the receiver's host.

Table 13. Connection Establishment Messages.

VARIABLES			
Name	Accessed by	Type	Purpose
Transmit	T2, T4	Boolean	Set to TRUE by T4 to indicate that a connection should be established
T_active	T2, T4	Boolean	Set to TRUE by T2 when the connection has been successful established with the receiver. Used to signal the start of the data transfer phase in the transmitter.
Fail	T2, T4	Boolean	Set to TRUE by T2 when the attempt to establish a connection failed because a responds to the *Con_req* was never received.
Accept	T2, T4	Boolean	Set to FALSE by T2 when parameters contained in *Con_ack* are unsatisfactory for the data transfer session.
R_active	T3	Boolean	Set to TRUE by R2 when the connection has been successful established with the transmitter. Used to signal the start of the data transfer phase in the receiver.
clock_tick (T2) clock_tick (R2)	T2 R2	periodic event	A timing event occurring at intervals of T_{in}.
delay (T2)	T2	counter	Used as an implicit timer for determining when T2 has waited a sufficient time period for a response to the previous *Con_req* message and that another one should be sent.
delay (R2)	R2	counter	Used as an implicit timer for determining when R2 has waited long enough for a response to the *Con_ack* message it sent.
attempts	T2	counter	Used as an implicit timer for determining when T2 has sent ample *Con_req* messages and waited long enough without a response from the receiver and the attempt to establish the connection should be aborted.

Table 14. Connection Establishment Phase Variables.

PROCEDURES		
Name	Parameter(s)	Function
Acceptable	message	Evaluates the connection parameters in the *Conn_ack* message. Returns true if the parameters are acceptable.
Dequeue	channel identifier[9]	Removes the data packet from the front of the indicated channel.
Empty	channel identifier	Returns true if the channel is empty.
Enqueue	message and channel identifier	Inserts the message (passed in as a parameter) into the indicated channel for transmission.
Evaluate	message	Processes the *Con_req* sent by the transmitter and determines the connection parameters to be sent in the *Con_ack* message.
Increment	counter variable	Increments the indicated counter variable.

Table 15. Connection Establishment Phase Procedures.

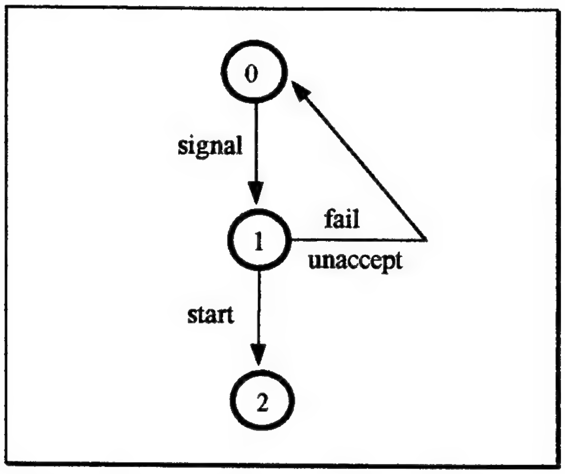

Figure 9. Machine T4 -- Host Interface

Transition	Predicate	Action
signal	*transmission request signal from host*	Transmit := TRUE;
fail	Fail = TRUE	Transmit := FALSE; *notify host of failure to connect*
unaccept	Accept = FALSE	*notify host of unacceptable connection*
start	T_active = TRUE	null

Table 16. Machine T4 Connection Establishment PAT.

[9] T_CHAN or R_CHAN.

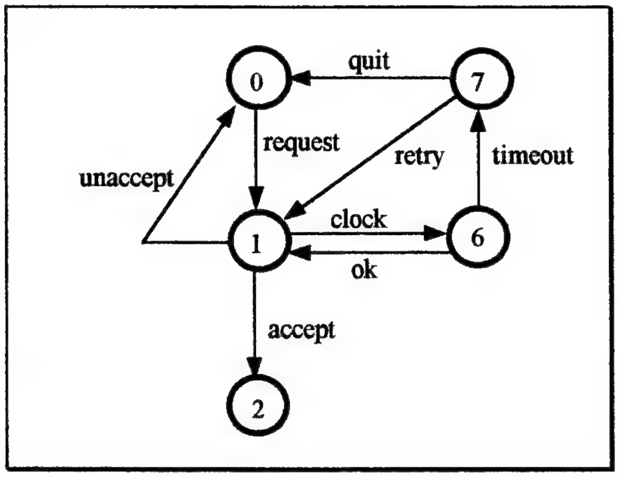

Figure 10. Machine T2 -- Transmitter Connection Management

Transition	Predicate	Action
request	Transmit = TRUE $\wedge$ Accept = TRUE $\wedge$ Fail = FALSE	**Enqueue**(*Conn_req*, T_CHAN);
accept	R_CHAN(front) = *Conn_ack* $\wedge$ **Acceptable** (R_CHAN(front))	T_active := TRUE; **Enqueue**(*Conn_conf*, T_CHAN); **Dequeue**(R_CHAN);
unaccept	R_CHAN(front) = *Conn_ack* $\wedge$ not (**Acceptable** (R_CHAN(front)))	Accept := FALSE: **Dequeue**(R_CHAN);
clock	**Empty**(R_CHAN) $\wedge$ *clock_tick*	**Increment**(delay);
ok	delay < reset	null
timeout	delay = reset	**Increment**(attempts); delay := 0;
retry	attempts < max_attempts	**Enqueue**(*Conn_req*, T_CHAN);
quit	attempts = max_attempts	Fail := TRUE;

Table 17. Machine T2 -- Connection Establishment PAT

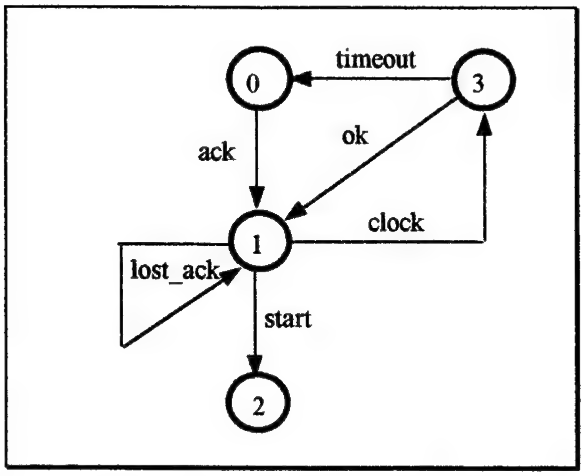

Figure 11. Machine R2 -- Receiver Connection Management.

Transition	Predicate	Action
ack	T_CHAN(front) = *Conn_req*	**Evaluate**(*Conn_req*); **Dequeue**(T_CHAN); **Enqueue**(*Conn_ack*, R_CHAN);
clock	**Empty**(T_CHAN) $\wedge$ *clock_tick*	**Increment**(delay);
ok	delay < reset	**Enqueue**(*Conn_ack*, R_CHAN);
timeout	delay = reset	null
start	T_CHAN(front) = *Conn_conf* $\vee$ T_CHAN(front) = *T_state* $\vee$ T_CHAN(front) = *Data*	R_active := TRUE; if T_CHAN(front) = *Conn_conf* then **Dequeue**(T_CHAN); end if;
lost_ack	T_CHAN(front) = *Conn_req*	**Dequeue**(T_CHAN); **Enqueue**(*Conn_ack*, R_CHAN);

Table 18. Machine R2 Connection Establishment PAT.

C. OPERATION

When the transmitter's host has data to send, the transmitter attempts to establish a connection between e itself and the receiver using a standard three-way handshake. This process was outlined in steps two through five of Table 10 in the previous chapter. A more comprehensive description of the actions of each machine is presented in this section. The operation of the connection establishment phase will first be explained when no error occurs. The names of variables are in bold type.

In the absence of errors, the actions of connection establishment phase occur as follows. T4 signals T2 that the transmitter's host has data to transfer to the receiver (**Transmit** set to TRUE). T2 checks **Transmit**, finds it value is TRUE, and sends a connection request message (*Con_req*) to the receiver. Information in *Con_req* specifies parameters desired by the transmitter for the connection. After processing the *Con_req* message, R2 respond with a connection acknowledgment message (*Con_ack*). *Con_ack* contains the connection parameters the receiver is able to accommodate. If the parameters returned by the receiver in the *Con_ack* message are acceptable to the transmitter, T2 sends a connection confirmation message (*Con_conf*) and signals the transmitter's other machines that the connection establishment phase was successful and to begin transferring data (**T_active** set to TRUE). Upon receiving the *Con_conf*, R2 signals the receiver's other machines to start the data transfer phase (**R_active** set to TRUE). The connection establishment phase of the protocol is complete and the transmitter and receiver begin the data transfer phase. If the transmitter finds the response of the receiver to its proposed connection parameters unacceptable, T2 quits its attempt to establish a connection and notifies T4 (**Accept** set to FALSE).

If the connection establishment phase was as uncomplicated as described above, its verification would be relatively simple. However, other situations may occur that must be taken into account. For example, if T2 fails to receive a *Con_ack* within a set time delay it sends another *Con_req* message. After a preset number of *Con_req* messages have been transmitted and no response has been received, T2 terminates the connection establishment phase and notifies T4 (**Fail** set to TRUE). Likewise, if R2 does not receive a response to its *Con_ack* within a set time delay the linkup routine in the receiver is terminated.

The coupling of concurrent actions and the possibility of problems due to failed machines or errors introduced by the network makes this apparently simple phase of the protocol more complex than expected.

D. PROPERTIES

The desired outcome of the connection establishment phase can be characterized intuitively as follows.

- If the transmitter's host has data to send one of two outcomes is acceptable.
    1. A connection is correctly established so that the data transfer can take place.
    2. If the connection cannot be established as required then the attempt is terminated and the transmitter's host is informed of the failure. Both the receiver and the transmitter should return to a ready condition.
- If data transfer has not been requested then a connection establishment is not attempted.

Specifically, the connection establishment phase must possess the safety and liveness properties listed in the table below. Additionally, the desired outcome is not guaranteed if deadlock is possible during this phase.

PROPERTIES		
Type	Label	Behavior Characterized
Safety	S1	If T2 signals that the connection has been successfully established (**T_active** = TRUE), then all variables relating to the connection establishment phase are consistent with this condition (**Accept** = TRUE and **Fail** = TRUE). It would be inappropriate for T4 to notify the host of a failure to connect while T1 is attempting to transmit data.
Safety	S2	When the connection establishment phase is completed successfully, both the transmitter and receiver are ready to commence the data transfer phase (**T_active** = TRUE and **R_active** = TRUE).
Safety	S3	If an attempt to establish the connection is unsuccessful, then either the response of the receiver was unacceptable (**Accept** = FALSE) or T2 failed to obtain a response from the receiver in the preset time limit (**Fail** = TRUE).
Liveness	L1	If T4 receives a transmission request from the transmitter's host then eventually either the connection is established (**T_active** = TRUE) or the attempt to establish the connection is unsuccessful and either **Accept** = FALSE or **Fail** = TRUE (see S3 above). In other words, the actions taken in the transmitter must produce an expected result.
Liveness	L2	If eventually the transmitter is ready for the data transfer phase (**T_active** = true) then the receiver will become ready to accept data (**R_active** = true) or the attempt to establish the connection is unsuccessful (again S3 above) or R2 times out and terminates its connection establishment effort. L2 differs from L1 in that it is based on the receiver -- actions taken in the receiver produce an expected result but only under the condition that the transmitter behaves properly.

Table 19. Connection Establishment Phase Properties.

Now that the properties have been determined the verification process can begin. Recall the task of verification is to show that these properties remain true in all reachable states of the connection establishment phase. Murphi is used to check for deadlock and the invariants of the above properties.

E. MURPHI DESCRIPTION

The first step for using Murphi to verify the connection establishment phase is to translate its SCM specification into a Murphi description. The SCM guarded transitions convert easily into Murphi's rules. Correctly expressing the properties is the more difficult task. The Murphi description of the connection establishment phase is displayed on the next three pages. Significant elements of each section of the description are discussed following the Murphi description.

/* Declarations */

```
Const
        reset_T2 : 2;           -- number of clock_ticks between Con_req retransmissions
        reset_R2 : 2;           -- number of clock_ticks before quitting
        max_attempts : 2;       -- number of times Con_req retransmitted before quitting

Type
        State_labels : 0..7;
        Message_type : Enum {None, Conn_req, Conn_ack, Conn_conf, T_state, Data};
        Counter_type : 0..2;  -- used for variables that increment or decrement

Var
        T2_state : State_labels;
        T4_state : State_labels;
        R2_state : State_labels;
        T_CHAN : Message_type;
        R_CHAN : Message_type;
        Host_T : Boolean; -- true if transmitter host has data to send
        Transmit : Boolean;
        T_active : Boolean;
        R_active : Boolean;
        Accept : Boolean;
        Fail : Boolean;
        delay_T : Counter_type;
        delay_R : Counter_type;
        attempts : Counter_type;
        R2_timeout : boolean;  -- true if transition timeout  taken by R2
```

/* Rules Section */

```
/* T4 transitions */

Rule "signal"
        T4_state = 0 & Host_T = true
==>
        T4_state := 1;
        Transmit := true;
End;

Rule "fail"
        T4_state = 1 &    Fail = true
==>
        T4_state := 0;
        Transmit := false;
        Host_T := false;
End;

Rule "unaccept_T4"
        T4_state = 1 & Accept = false
==>
        T4_state := 0;
        Host_T := false;
End;

Rule "start_T4"
         T4_state = 1 & T_active = true
==>
    T4_state := 2;
End;
```

```
/* T2 transitions */

Rule "request"
          T2_state = 0 & Transmit = true &
          Accept = true & Fail = false
==>
          T2_state := 1;
          T_CHAN := Conn_req;
End;

Ruleset P_acceptable : Boolean Do
Rule "accept"
          T2_state = 1 & R_CHAN = Conn_ack
& P_acceptable = true
==>
          T2_state := 2;
          T_active := true;
          T_CHAN := Conn_conf;
          R_CHAN := None;
End;

Rule "unaccept_T2"
          T2_state = 1 & R_CHAN = Conn_ack
& P_acceptable = false
==>
          T2_state := 0;
          Accept := false;
          R_CHAN := None;
End;
Endruleset;

Rule "clock_T2"
          T2_state = 1 & R_CHAN = None
==>
          T2_state := 6;
          delay_T := delay_T + 1;
End;

Rule "ok_T2"
          T2_state = 6 & delay_T < reset_T2
==>
          T2_state := 1;
End;

Rule "timeout_T2"
          T2_state = 6 & delay_T = reset_T2
==>
          T2_state := 7;
          attempts := attempts + 1;
          delay_T := 0;
End;

Rule "retry"
          T2_state = 7 & attempts <
max_attempts
==>
          T2_state := 1;
          T_CHAN := Conn_req;
End;

Rule "quit"
          T2_state = 7 & attempts =
max_attempts
==>
          T2_state := 0;
          Fail := true;
End;


/* R2 transitions */

Rule "ack"
          R2_state = 0 & T_CHAN = Conn_req
==>
          R2_state := 1;
          T_CHAN := None;
      R_CHAN := Conn_ack;
End;

Rule "clock_R2"
          R2_state = 1 & T_CHAN = None
==>
          R2_state := 3;
          delay_R := delay_R + 1;
End;

Rule "ok_R2"
          R2_state = 3 & delay_R < reset_R2
==>
          R2_state := 1;
          R_CHAN := Conn_ack;
End;

Rule "timeout_R2"
          R2_state = 3 & delay_R = reset_R2
==>
          R2_state := 0;
      delay_R := 0;
      R2_timeout := true;
End;
```

```
Rule "start_R2"
        R2_state = 1 &
        (T_CHAN = Conn_conf | T_CHAN =
T_state | T_CHAN = Data)
==>
        R2_state := 2;
        R_active := true;
        if T_CHAN = Conn_conf then
                T_CHAN := None;
        endif;
End;

Rule "lost_ack"
        R2_state = 1 & T_CHAN = Conn_req
==>
        R2_state := 1;
        T_CHAN := None;
        R_CHAN := Conn_ack;
End;
```

/* Initialization section */

```
Startstate
        Host_T := true;
        T2_state := 0;
        T4_state := 0;
        R2_state := 0;
        T_CHAN := None;
        R_CHAN := None;
        Transmit := false;
        T_active := false;
        R_active := false;
        Accept := true;
        Fail := false;
        delay_T := 0;
        delay_R := 0;
        attempts := 0;
        R2_timeout := false;
End;
```

/* Properties */

```
Invariant "-- consistent conditions at connection establishment --"
        T_active = true → (Accept = true & Fail = false);

Invariant "-- transmitter and receiver ready at end of phase --"
        R_active = true  →  T_active = true;

Invariant "-- not both fail and unaccept --"
        !(Fail = true & Accept = false);

Liveness " -- connection established if desired --"
        Always Transmit = true  →  Eventually (( T_active = true) |
        (Fail = true | Accept = false));

Liveness "-- xmitter ready followed by rcvr ready --"
        Eventually Always (T_active = true & R_active = true) | Fail = true |
        Accept = false | R2_timeout = true;
```

1. Declarations

First three constants are declared. The value of "reset_T2" limits the number of times the "clock $\rightarrow$ ok" loop (state 1 to state 6 back to state 1, etc.) is executed by T2 prior to retransmitting a *Con_req* message. Since **clock_tick** occurs at an interval of T_{in}, this loop serves as a retransmission timer T2. The constant "max_attempts" fixes the number retransmissions of *Con_req* conducted prior to giving up the attempt to establish the connection. A similar limit on the number of times *Con_ack* is retransmitted is found in R_2 with the constant "reset_R2".

Next, three data-types are defined. The type "State_labels" specifies an integer subrange which ranges over the reachable states of T2, T4, and R2. The next type declaration is an enumeration type, called "Message_type". Its domain includes all messages that could be sent during the connection establishment phase. "None" indicates that the channel is empty.

The variable required for the connection establishment phase comprise the final part of the declaration section. They have already been explained in Table 13. In this description, **T_CHAN** and **R_CHAN** are implemented as scalars. A queue is unnecessary since the ordering of message in the channel has no impact on the operation of the connection establishment phase. Also, any time delay corresponding to the network's propagation delay is ignored.

The introduction of a variable to implement a periodic clock event is not required. The presence or absence of a variable that alternates between two values has no impact on the verification of the connection establishment phase.

2. Rules

The rules are grouped by the machine to which they apply. Note the correspondence between each rule and the associated predicate and action of the guarded transitions listed in the PAT's. The guard for all rules involve the current state of the associated machine and the value of one or more variables. For most rules, the actions and the conditions under which the their bodies are executed is clear and no explanation is required.

Slightly more complicated is the ruleset that is part of T2's description. The two rules "accept" and "unaccept_T2" comprise the body of the ruleset. The ruleset, causes the value of quantifier "P_acceptable" to alternate between TRUE and FALSE. This allows the behavior of the connection establishment phase to be examined when the receiver responds with connection parameters acceptable to the transmitter and also when unacceptable parameters are sent.

3. Startstate

Variable initialization is as expected. The machines all start from state zero. The channels are empty. The values of Boolean variables reflect an idle transmitter and receiver. All counters used for the implicit timer are set to zero.

4. Invariants

The invariants and liveness constructs correspond to the properties defined in Table 18. The second invariant's formula, **R_active** = TRUE $\rightarrow$ **T_active** = TRUE, states "if the receiver is ready for the data transfer phase then the transmitter is also. This differs slightly from what is written in Table 18 (**T_active** = TRUE and **R_active** = TRUE). The modification is necessary to prevent a false invariant violation. During execution, **T_active** is set to TRUE prior to **R_active** being set to TRUE. (**R_active** is not set to TRUE until after the message sent by the transmitter in respond to a *Con_ack*, is received by R2). Since they are not both set to TRUE simultaneously, the formula **T_active** = TRUE & **R_active** = TRUE is not invariant.

The first liveness formula, as implemented in Murphi is equivalent to:

$$\text{ALWAYS}\ (p \rightarrow \text{EVENTUALLY}\ q).$$

The second liveness formula, of the form EVENTUALLY ALWAYS (p), means at some point p is true and remains true from then on.

F. RESULTS

Analysis of results obtained from Murphi indicates the connection establishment phase of SNR functions properly. However, two interesting circumstances were observed.

The first is a condition flagged by Murphi as a deadlocked state. If near the end of the connection establishment phase, R2 times out before it receives a *Con_conf* message, a data packet, or a state packet from the transmitter, and after T2 has set **T_active** to TRUE, the protocol ends up in a condition with the transmitter ready to send data but the receiver has quit the connection.[10] This appears to be a liveness violation, however, when the connection establishment phase and the data transfer phase are taken together, deadlock is avoided The protocol eventually returns to the initial state since the transmitter will timeout and disconnect when receiver state packets are not received (occurs in the data transfer phase). The sequence of events pertaining to this situation is given in the table below.

Event Description	State of Machines		
	T4	T2	R2
A *Con_ack* message is sent by R2.	1	1	1
R2 increments **delay**.			3
delay is less than **reset** so another *Con_ack* message is sent by R2.			1
Con_ack received at T2 and the connection parameters it contains are acceptable to the transmitter. **T_active** is set to TRUE. A *Con_conf* message is sent by T2.		2	
R2 increments **delay**			3
delay is less than **reset** so another *Con_ack* message is sent by R2			1
R2 continues to execute the "clock" "ok" transition. **delay** is incremented each cycle.			1, 3, 1, ...
The value of **T_active** checked by T4 and found to by TRUE. The transmitter begins the data transfer phase and sending data packets to the receiver.	2		
delay = **reset** so the "timeout" transition is taken by R2.			0
After a period of time the transmitter will terminate the connection since T3 will never receive a control pack from the receiver.	0	0	

Table 20. Events Leading to Unexpected Condition.

[10] This problem is inherit to the three way handshake and is known as the "three army problem".

Also of interest is the need for a conjunction of three variables in the guard of transition "request" of machine T2 to prevent livelock. If **Transmit** = TRUE was the only component of the predicate for "request", then once the host signaled it had data to transfer, the rule would be enabled until **Transmit** was reset by T4. However, if T4 was never again given an opportunity to execute its actions (T2 and T4 running on a single CPU and starvation of T4 occurs) then the "request" rule could fire infinitely often. Including **Accept** = TRUE and **Fail** = FALSE with **Transmit** = TRUE eliminates the possibility of livelock.

VII. VERIFICATION -- FLOW CONTROL MODE OF SNR

A. INTRODUCTION

The correctness of SNR's connection establishment phase was examined in the previous chapter. The next step is verifying the protocol's data transfer phase. Instead of attempting to investigate the data transfer phase in its entirety, a modular approach is employed. Recall, in SNR the data exchange can occur without flow or error control (Mode 0), with only flow control (Mode 1), or with both error and flow control (Mode 2). Even though Mode 0 is the least complicated of the three modes, and therefore its verification is the next logical step, its explicit verification is skipped. Mode 1 essentially includes all of the states and actions of Mode 0. (The only difference is in machine R1. In Mode 1, R1 changes from state 2 to state 3 and then back to state 1, while in Mode 0 R1 changes directly back to state 1 from state 2.) This chapter describes the verification of SNR's data transfer phase operating with flow control only. The verification is accomplished with the assistance of the Murphi Verification System. Additionally, state space explosion, as it applies to the Murphi description of SNR's Mode 1, is explored. It is important to determine whether state space explosion will prevent the full verification of SNR.

This chapter follows a format similar to the previous chapter. The architecture of SNR applicable to flow control is addressed first, followed by a description of actions in Mode 1. Next the safety property applicable to flow control is explored and then the Murphi description is presented. Finally verification results and state space explosion are discussed.

B. MACHINE DIAGRAMS -- MODE 1

No new material is present in this section. Chapter V, serves as the framework for this chapter.[11] The packet types, variables, connection parameters, etc., applicable to Mode 1 are the same as in Mode 2 and have already been discussed in Chapter V. Only T1, T2, R1 and R3 perform functions in Mode 1. Presented below are extracts from the diagrams and tables of Section V.D.6 for these four machines. Just those states and transitions involved in flow control are shown.

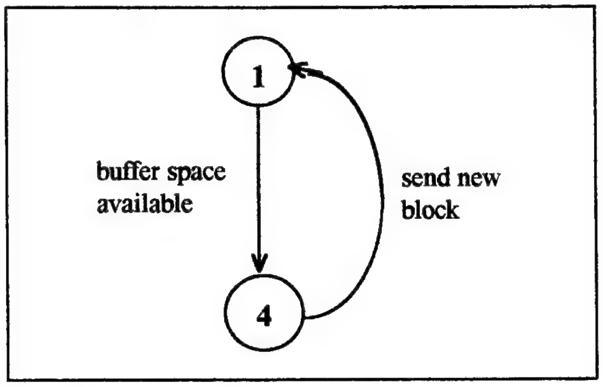

Figure 12 Machine T1 State Diagram

TRANSITIONS	EXPLANATION
$1 \rightarrow 4$	Occurs when information at the transmitter indicates there is sufficient space in the receiver's buffer for a block of data packets (*buffer_available* > 0).
$4 \rightarrow 1$	Occurs after the transmitter has: sent a new block; updated the table of outstanding blocks (*LUP*); and set *busy* to true.

Table 21 Transitions for Machine T1.

[11] A principal goal of this thesis is verifying the SNR protocol as introduced in [NRS90]. Therefore the specification given in [McAr92] and as refined in [Tipi93] does not play a primary role in the verification of the data transfer phase

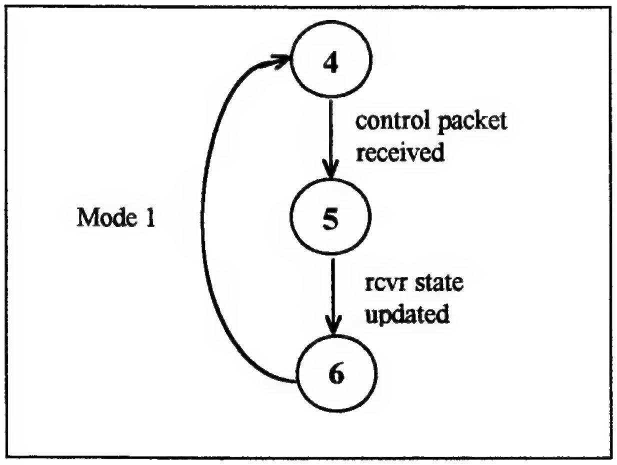

Figure 13 Machine T2 State Diagram.

TRANSITION	EXPLANATION
4 → 5	Occurs if a control packet is received from the receiver.
5 → 6	Occurs after updating the receiver's state information maintained at the transmitter and the disconnect counter (*scount*) has been reset.
6 → 4	Occurs if Mode 1 are being used

Table 22. Transitions for Machine T2.

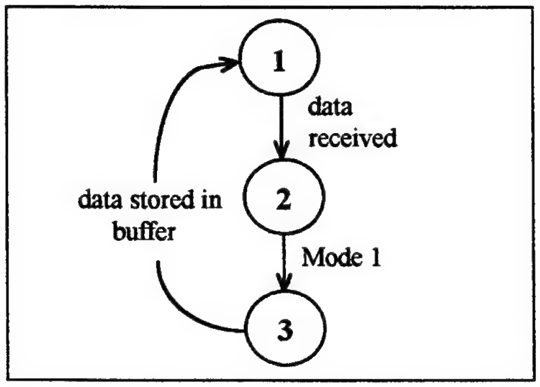

Figure 14. Machine R1 State Diagram.

TRANSITIONS	EXPLANATION
1 → 2	Occurs if a data packet is received from the transmitter.
2 → 3	Occurs if Mode 1 is being used.
3 → 1	Occurs after data is stored in the receiver's buffer.

Table 23. Transitions for Machine R1.

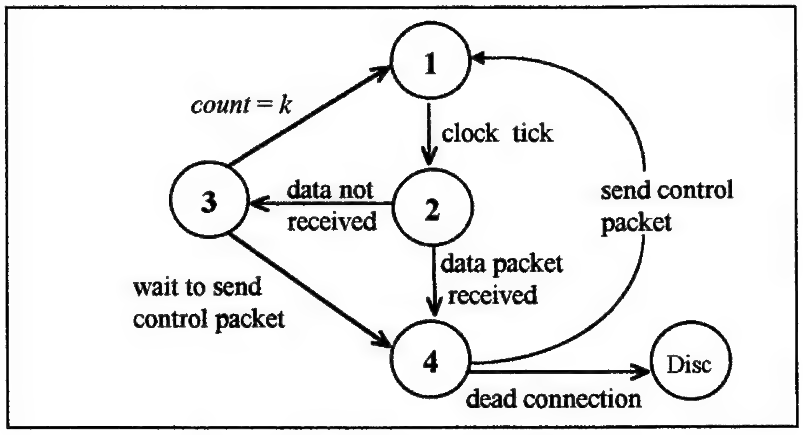

Figure 15. Machine R3 State Diagram.

TRANSITIONS	EXPLANATION
1 → 2	Occurs if event *clock_tick* detected and after *scount* incremented.
2 → 3	Occurs if a new data packet has **not** been received and after *count* incremented.
2 → 4	Occurs if a new data packet received and after *count* incremented.
3 → 1	Occurs if it is not yet time to send a control packet (*count* < *k*).
3 → 4	Occurs if *count* = *k* and after *k* has been modified to reduce the transmission rate of receiver control packets.
4 → 1	Occurs after a control packet is sent, and after *busy* and, *count*, are reset.
4 → *Disc*	Occurs if the receiver has **not** received a control packet from the transmitter in the expected interval (*scount* reached predetermined value).

Table 24. Transitions for Machine R3.

C. OPERATION

The purpose of flow control in SNR is to permit the transmitter to send as many data packets as possible without overflowing the receiver's buffer. This is accomplished by regulating the transmission of data packets using information about the receiver as know at the transmitter. In SNR flow control is based on blocks of data packets not individual packets.

After the connection establishment phase is complete the protocol enters the data transfer phase. Below are the basic operations performed during a data transfer session utilizing Mode 1 .

- T1 transmits blocks of data packets until the preset limit on the capacity of the receiver's buffer is reached.
- R1 stores the incoming data packets in its buffer. Packets are removed from the buffer by the receiver's host. The status of buffer space (value of variable buffer_available) is updated as new packets are inserted and the host removes packets.
- At the appropriate interval, R3 sends receiver state information to T2. This information is used to update the state of the receiver's buffer (as known by the transmitter).
- T1 temporally halts transmission of data packets when its information indicates the receiver's buffer will be full when all of the data packets it has sent arrive at the receiver. T1 resumes sending data packets when state information from the receiver indicates buffer space in once again available.
- Control packets and blocks of data packets continue to be exchanged until the entire message has been acknowledged by the receiver.
- R3 terminates the connection if data packet is not received within the required time limit.

The basic operations performed in Mode 1 are similar to those explained in Chapter V for SNR transferring data using Mode 2. The primary difference is that in Mode 1 errors are ignored. As a result, retransmission of data packets and all the associated processing needed to accomplish retransmission is omitted. There are six areas impacted significantly.

1. In Mode 1, T1 sends data packets as along as ($buffer\_available_{transmitter} > 0$) is true. In Mode 2 new data packets are transmitted when the retransmission of a block of data packets is not required and the predicate ($buffer\_available_{transmitter} - NOU > 0$) is true.
2. The retransmission table (*LUP*) is not maintained in Mode 1.
3. In Mode 1, T3 remains in state 1. Therefore transmitter state packets are not sent to the receiver

4. R1 stores, without processing, data packets in the buffer for the host to retrieve. It does no processing of the data packets since errors are ignored.
5. R2 remains in state 2 since a control packet is never received from the transmitter (because of number 3 above).
6. Since data packets are not processed by the receiver (see number 4 above), no meaningful status information other than buffer space available can be sent in control packets by R3.

The differences in these six areas simplify considerable, as compared to Mode 2, the behavior and the Murphi description for Mode 1.

D. PROPERTIES

The primary safety property for SNR operating with only flow control is, the receiver's buffer must not overflow. That is the condition ($buffer\_available_{receiver} \geq 0$) must always be true.

E. MURPHI DESCRIPTION

The Murphi description for SNR's data transfer phase operating in Mode 1 was developed with three goals in mind.

1. The description must correctly characterize the behavior of Mode 1.
2. The description should serve as the groundwork for SNR's data transfer phase operating in any mode (allow scaling up to Mode 2).
3. The description should be as simple as possible (to enhance its understandably). Only those actions specifically required for flow control should be implemented and the number of variables kept to the absolute minimum (to reduce the size of the state space).

A fact important to achieving goal number three above is: flow control is accomplish in SNR by managing blocks of data packets. As a result, the state space is reduced since the description can be based on data blocks and the variables and data structures needed for tracking individual data packets can be eliminated.

Displayed on the next six pages is the Murphi description for Mode 1. At this point the reader has been exposed to numerous descriptions written in the Murphi

Descriptive Language and much of this description should be familiar. Therefore only a few points specific to this particular description are covered below.

The name of each rule has been formatted to facilitate understanding the purpose of the rule as follows:

machine identification - description of guard or action for rule - current state of machine.

For example: "R1 - receive data packet - rs1" indicates this is a rule for machine R1, the body of the rule is executed when a data packet is received, and R1 must be in state 1 (rs1) for the rule to fire.

In this description **T_CHAN** and **R_CHAN** are implemented as circular arrays. Each element of **T_CHAN** can contain either a block of data packets or is empty. Likewise each element of **R_CHAN** contains a receiver control packet or is empty. Again, the network's propagation delay is ignored. (Note the size of the arrays is only two elements. This is because these data structures contribute to the overall state space. Increasing their size significantly impacts the state space.)

The description also contains **bold type** and *italicized* code. When the **bold type** code is removed and the *italicized* code added, an alternate implementation based on the SCM specification from [McAr92] is created. The specification as given in [McAr92] is examined because the design of SNR, with flow control based strictly on the variable buffer_available, is flawed. See Section F below for a full explanation.

```
--*******************************************************************
--                          Declarations
--*******************************************************************

Const
  chan_cap: 2;             -- channel capacity in blocks
  message_size: 3;         -- number of blocks in message
  max_time_interval: 8;    -- maximum change in k
  rcvr_buffer_size: 2;     -- size of rcvr buffer
  scount_lim: 12;          -- upper bound on the value of scount
                           -- connection terminated if scount reaches this value
```

```
Type
  counter_type: 0..30;
  time_interval_type: 0.. max_time_interval;
  buffer_type: -1..rcvr_buffer_size;
  block_seq_type: 0..message_size;            -- basic counter type for blocks and block sequence numbers

  T_states_type: Enum {ts1, ts2, ts3, ts4, ts5, ts6};
  R_states_type: Enum {rs1, rs2, rs3, rs4};
  chan_slot: 0..(chan_cap - 1);
  T_packet_type: Enum {none_T, datapac};
  R_packet_type: Enum {none_R, conpac};

  T_Packet_record:
    Record
      packet_kind: T_packet_type;                    -- kinds of packet
      -- (NOU) seq_num: block_seq_type;  -- packet sequence number
    End;

  T_CHAN_type: array [chan_slot] of T_Packet_record;

  R_Packet_record:
    Record
      packet_kind: R_packet_type;   -- kind of packet
      -- (NOU) LW_R: block_seq_type;        -- below LW_R all blocks received
      buffer_avail: buffer_type;  -- rcvr buffer status
    End;

  R_CHAN_type: array [chan_slot] of R_Packet_record;

Var
  T1_state: T_states_type;
  T2_state: T_states_type;
  R1_state: R_states_type;
  R3_state: R_states_type;

  T_CHAN: T_CHAN_type;          -- communication channel from xtmr to rcvr
  R_CHAN: R_CHAN_type;          -- communication channel from rcvr to xtmr
  xtmr_end_TC: chan_slot;       -- transmitter end of T_CHAN
  rcvr_end_TC: chan_slot;       -- receiver end of T_CHAN
  xtmr_end_RC: chan_slot;       -- transmitter end of R_CHAN
  rcvr_end_RC: chan_slot;       -- receiver end of R_CHAN

  k_T: time_interval_type;      -- value of time interval at xtmr
  k_R: time_interval_type;      -- value of time interval at rcvr
  latest_Tpacket: T_Packet_record;          -- block at rcvr from xtmr
  latest_Rpacket: R_Packet_record;           -- control packet at xtmr from rcvr
   blk_seq_num: block_seq_type;             -- seq num for entire block
  OUTBUF: block_seq_type;       -- contains message to be sent
  buffer_avail: buffer_type;    -- buffer space available in rcvr
  buffer_avail_T: buffer_type;  -- value at xtmr
  -- (NOU) NOU: block_seq_type; -- number of blocks outstanding
  UW_T: block_seq_type;          -- blk seq # < UW_T have all been sent
  LW_R: block_seq_type;          -- blk seq # < LW_R have all been received
  LW_T: block_seq_type;          -- value of LW_T at xtmr

  T_busy: Boolean;               -- status of sending data packets
  R_busy: Boolean;               -- status of receiving data packets
  scount_R: counter_type;        -- counter for disconnect if no flow
  count_R: time_interval_type;   -- counter for adjusting k_R


--++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
--                 PROCEDURES
--++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* ________ send_block _________ */

Procedure send_block();  -- places blocks worth of data packets in T_CHAN

Var
  next_Tpacket: T_Packet_record;  -- next packet at xtmr to send

Begin

  blk_seq_num := blk_seq_num + 1;
  next_Tpacket.packet_kind := datapac;
  -- (NOU) next_Tpacket.seq_num := blk_seq_num;

  T_CHAN[xtmr_end_TC] := next_Tpacket;
  xtmr_end_TC := (xtmr_end_TC + 1) % chan_cap;

  UW_T := blk_seq_num;
  -- (NOU) NOU := NOU + 1;
  OUTBUF := OUTBUF - 1;

  buffer_avail_T := buffer_avail_T - 1;

End;  -- send_block


/* ________ receive_conpac _________ */

Procedure receive_conpac();  -- xtmr receives control packet form rcvr

Begin

  latest_Rpacket := R_CHAN[xtmr_end_RC];
  R_CHAN[xtmr_end_RC].packet_kind := none_R;
  xtmr_end_RC := (xtmr_end_RC + 1)  % chan_cap;  -- consumes packet

End;  -- receive_conpac
```

```
/* ________ receive_block _________ */

Procedure receive_block();  -- rcvr receives an entire block of data packets

Begin

  latest_Tpacket := T_CHAN[rcvr_end_TC];
  T_CHAN[rcvr_end_TC].packet_kind := none_T;
  rcvr_end_TC := (rcvr_end_TC + 1)  % chan_cap;

End;  -- receive_block

/* ________ store_block _________ */

Procedure store_block();  -- makes block available to rcvr host

Begin

  buffer_avail := buffer_avail - 1;

End;  -- store_block

/* ________ send_control_packet_R _________ */

Procedure send_control_packet_R();  -- sends rcvr control packet

Var
  next_Rpacket: R_Packet_record;   -- next packet at rcvr to send

Begin
  -- load data in packet
  next_Rpacket.packet_kind := conpac;
  -- (NOU) next_Rpacket.LW_R := LW_R;
  next_Rpacket.buffer_avail := buffer_avail;

  -- place packet in channel
  R_CHAN[rcvr_end_RC] := next_Rpacket;
  rcvr_end_RC:= (rcvr_end_RC + 1) % chan_cap;

End;  -- send_control_packet_R
```

```
--***********************************************************
--                RULES
--***********************************************************

/* ........... T1 transitions ........... */

Rule "T1 - transmit possible - ts1"

  (T1_state = ts1) & (buffer_avail_T > 0)
            -- (NOU) ((T_buffer_avail - NOU)
> 0)
==>
  T1_state := ts4;
End;

Rule "T1 - transmit block - ts4"

  (T1_state = ts4) & (OUTBUF > 0)
==>
  send_block();
  T_busy := true;
  T1_state := ts1;
End;

/* ....... T2 transitions ......... */

Rule "T2 - receive rcvr state info - ts4"

  (T2_state = ts4) &
(R_CHAN[xtmr_end_RC].packet_kind =
conpac)
==>
  receive_conpac();
  T2_state := ts5;
End;

Rule "T2 - update info about rcvr - ts5"

  (T2_state = ts5)
==>
  -- update information at xtmr
  -- (NOU) LW_T := latest_Rpacket.LW_R;
  buffer_avail_T := latest_Rpacket.buffer_avail;

  T2_state := ts6;
End;

Rule "T2 - go back to ts4 - ts6"

  (T2_state = ts6)
==>
  T2_state := ts4;
End;

/* ....... R1 transitions ....... */

Rule "R1 - receive data packet - rs1"

  (R1_state = rs1) &
(T_CHAN[rcvr_end_RC].packet_kind =
datapac)
==>
  receive_block();
  R_busy := true;
  R1_state := rs2;
End;

Rule "R1 - process data packet -  rs2"

  (R1_state = rs2)
==>
  -- (NOU) LW_R := latest_Tpacket.seq_num;

  R1_state := rs3;
End;

Rule "R1 - store data packet -  rs3"

  (R1_state = rs3)  -- (NOU) &
   -- (latest_Tpacket.seq_num > LW_R - 1)
==>
  store_block();
  R1_state := rs1;
End;

/* ....... rcvr host ...... */
-- included to simulate action of the  rcvr's host
Rule "remove packet from buffer"

  buffer_avail < rcvr_buffer_size
==>
    buffer_avail := buffer_avail + 1;
end;
```

```
/* ........ R3 transitions ........ */

Rule "R3 - clock_tick -  rs1"

  (R3_state = rs1)
==>
  scount_R := scount_R + 1;
  R3_state := rs2;
End;


Rule "R3 - not busy -  rs2"

  (R3_state = rs2) & (R_busy = false)
==>
  count_R := count_R + 1;
  R3_state := rs3;
End;


Rule "R3 - busy -  rs2"

  (R3_state = rs2) & (R_busy = true)
==>
  k_R := 1;
  R3_state := rs4;
End;


Rule "R3 - wait (count_R < k_R) -  rs3"

  (R3_state = rs3) & (count_R < k_R)
==>
  R3_state := rs1;
End;


Rule "R3 - modify k_R -  rs3"

  (R3_state = rs3) & (count_R = k_R)
==>
  If k_R < max_time_interval Then
    k_R := k_R * 2;;
  Endif;

  R3_state := rs4;
End;


Rule "R3 - send rcvr state -  rs4"

  (R3_state = rs4) & (scount_R < scount_lim)
==>
  send_control_packet_R();
  count_R := 0;
  R_busy := false;
  R3_state := rs1;
End;


Rule "R3 - disconnect -  rs4"

  (R3_state = rs4) & (scount_R = scount_lim)
==>
  error "disconnect";

End;
```

```
--*************************************************
                          Startstate
--*************************************************

  T1_state := ts1;
  T2_state := ts4;
  R1_state := rs1;
  R3_state := rs1;

For cs: chan_slot Do   -- fill channels with empty packets
     T_CHAN[cs].packet_kind := none_T;
     -- (NOU) T_CHAN[cs].seq_num := 0;
     R_CHAN[cs].packet_kind := none_R;
     -- (NOU) R_CHAN[cs].LW_R := 0;
     R_CHAN[cs].buffer_avail := rcvr_buffer_size;
  Endfor;

  xtmr_end_TC := 0;
  rcvr_end_TC := 0;
  xtmr_end_RC := 0;
  rcvr_end_RC := 0;
  latest_Tpacket.packet_kind := none_T;
  -- (NOU) latest_Tpacket.seq_num := 0;
  latest_Rpacket.packet_kind := none_R;
  -- (NOU) latest_Rpacket.LW_R := 0;
  latest_Rpacket.buffer_avail := rcvr_buffer_size;
  k_T := 1;
  k_R := 1;
  blk_seq_num := 0;
  OUTBUF :=  message_size;
  buffer_avail := rcvr_buffer_size;
  buffer_avail_T := rcvr_buffer_size;
  -- (NOU) NOU := 0;
  UW_T := 0;
  LW_R := 0;
  LW_T := 0;
  T_busy := false;
  R_busy := false;
  scount_R := 0;
  count_R := 0;
End;

--*****************************************************
--                        Invariant
--*****************************************************

Invariant "-- no buffer overflow  --"
  buffer_avail > -1;   -- overflow occurs when buffer space is zero
```

F. RESULTS

The data transfer phase of SNR operating with flow control only (Mode 1) does not behave as desired. Two problems were discovered.

1. The receiver's buffer can overflow.
2. The improper termination of the connection can occur.

Mode 0 also exhibits problem number two. State space explosion did not prevent verification of Mode 1 using the description based on [NRS90] or based on [McAr92]. However, the size of the channels and message must be severely restricted to avoid significantly increasing the state space. The errors discovered in the design of Mode 1 and state space explosion are discussed in greater detail below.

1. Buffer Overflow

The first problem arises because the variable $\text{buffer\_available}_{\text{transmitter}}$ used to prevent does buffer overflow does not reflect the **current** status of the buffer space available at the receiver. In Mode 1, T1 checks that the value of $\text{buffer\_available}_{\text{transmitter}}$ is greater than zero 0, sends a new block of data packets and then decrements $\text{buffer\_available}_{\text{transmitter}}$. Data blocks continue to be transmitted by T1 until $\text{buffer\_available}_{\text{transmitter}}$ reaches zero. The problem occurs when R3 sends a control packet to T2 just prior to R1 storing some number of data packets in the receiver's buffer. The control packet sent by R3 in this situation contains a value for the space available in the buffer that reflects space in the buffer subsequent to R1 storing data packets. The value of $\text{buffer\_available}_{\text{transmitter}}$ is updated by T2 using information that indicated there is more space in the buffer than actually exist. So, if $\text{buffer\_available}_{\text{transmitter}}$ was zero it will be set to a value larger than the actual space available in the buffer. T1 checks and finds $\text{buffer\_available}_{\text{transmitter}} > 0$, so T1 resumes sending blocks of data packets. If the host linked to the receiver has not yet removed any packets from the buffer prior to the arrival of the latest batch of data blocks, the capacity of the receiver's buffer will be exceed. The following example uses an arbitrary sized message and buffer to illustrates how buffer overflow arises:

Assume.

The transmitter has 100 blocks to send.
The receiver's buffer capacity is 10 blocks.

Initial Conditions.

Transmitter		Receiver	
Blocks to Transmit	100	Blocks Received	0
buffer_available$_{transmitter}$	10	buffer_available$_{receiver}$	10

T1 sends 10 blocks of data packets.

Transmitter		Receiver	
Blocks to Transmit	90	Blocks Received	0
buffer_available$_{transmitter}$	0	buffer_available$_{receiver}$	10

R3 sends a control packet with a value of 10 in the buffer_available field.

Transmitter		Receiver	
Blocks to Transmit	90	Blocks Received	0
buffer_available$_{transmitter}$	0	buffer_available$_{receiver}$	10

The data blocks arrive at R1 and are place in the buffer.

Transmitter		Receiver	
Blocks to Transmit	90	Blocks Received	10
buffer_available$_{transmitter}$	0	buffer_available$_{receiver}$	0

The receiver's host removes 3 blocks from the buffer.

Transmitter		Receiver	
Blocks to Transmit	90	Blocks Received	10
buffer_available$_{transmitter}$	0	buffer_available$_{receiver}$	3

T2 receives the control packets and updates buffer_available$_{transmitter}$.

Transmitter		Receiver	
Blocks to Transmit	90	Blocks Received	10
buffer_available$_{transmitter}$	10	buffer_available$_{receiver}$	3

T1 sends 10 more blocks.

Transmitter		Receiver	
Blocks to Transmit	80	Blocks Received	10
buffer_available$_{transmitter}$	0	buffer_available$_{receiver}$	3

The receiver's host removes 3 more blocks from the buffer.

Transmitter		Receiver	
Blocks to Transmit	80	Blocks Received	10
buffer_available$_{transmitter}$	0	buffer_available$_{receiver}$	6

R1 receives 10 data blocks, 6 are placed in the buffer. The remaining 4 blocks overflow the buffer.

The trace of the execution path produced by Murphi for this design error is provided in Appendix B.

To prevent buffer overflow the condition on the transition from state 1 to state 4 in machine T1 must be changed. If the predicate (buffer_available$_{transmitter}$ - NOU > 0) is used in place of (buffer_available$_{transmitter}$ > 0), the overflow problem is eliminated. In addition to this change in T1, the sequence number of the most recent blocks processed by R1 must be send to T2 by R3 in the receiver control packet. This information is then used to update NOU. Removing the **bold type faced** code in the Murphi description and adding the *italicized* code produces a description that does not exhibit the buffer over flow problem. This alternate description for SNR's Mode 1 comes from the specification given in [McAr92].

2. Undesired Disconnection

The second problem, undesired disconnection, occurs because *scount$_{receiver}$* is never reset in Mode 1. R3 increments *scount$_{receiver}$* each time the transition from state 1 to state 2 is taken. The value of *scount$_{receiver}$* is checked against its upper bound (*scount_lim*) in state 4. If *scount$_{receiver}$* equals *scount_lim* then the connection is terminated at the receiver, otherwise the data exchange continues. The variable scount$_{receiver}$ is only reset to zero by machine R2 when a control packet arrives at the receiver from T3. However in Mode 1, T3 never sends a control packet so *scount*$_{receiver}$ is never reset to zero. Therefore unless the message is very short, *scount*$_{receiver}$ will reach *scount_lim* and R3 will terminate the connection prematurely.

This problem is masked by buffer overflow when using the Murphi description based on specification from [NRS90]. The alternate description produced from the specification in [McAr92] does not cause the receiver's buffer to overflow. As a result, the error in the design of the receiver's disconnect timer was discovered.

3. State Space Explosion

State space explosion was avoided in the verification of SNR's data transfer phase operating using flow control only by using a very small message, short channels and a tiny buffer. The buffer overflow problems in the description based on [NRS90], was detected by Murphi after 19,652 states had been explored. The *scount* reset problem encounter when the alternate description (based on specification in [McAr92]) was used, occurred after examining 442,369 states. Changing the size of the channel from two to three, in the alternate description, resulted in over 671,000 states examined prior to detecting the *scount* error. Adding the extra states and variables required to fully describe SNR's data transfer phase operating with both flow control and error will significantly increase the number of states generated by Murphi.

VIII. CONCLUSIONS AND RECOMMENDATION

In this thesis the correctness of SNR's design was examined. Key properties of SNR's connection establishment phase and data transfer phase operating in Mode 1 were identified and verified. A summary of the verification results is present in the first section of this chapter. The second section discusses the feasibility of using the Murphi Verification System for verifying communication protocols. The final section provides recommendations for completing the verification of SNR and for enhancing Murphi's capabilities with respect to protocol verification.

A. SUMMARY -- VERIFICATION OF SNR

The design of SNR as presented in [NRS90] appears to contain inconsistencies. Two problems in the actions of the protocol's data transfer phase operating with flow control only (Mode 1) were detected by Murphi. The first is a violation of the key property of flow control -- buffer overflow must not occur. The second is a violation of the basic liveness property applicable to all protocols -- the message is eventually delivered. Both problems are the result of improper coordination between the transmitter and the receiver.

1. The receiver's buffer can overflow. The strategy expected to halt the transmission of blocks of data packets prior to exceeding the capacity of receiver's buffer, does not function as intended. The scheme as specified fails to take into account that there may be data blocks in transit (sent by the transmitter but have not yet arrived at the receiver). In this situation each individual machine functions properly but it is the coordination between the transmitter and receiver that is flawed.

2. The network connection between the transmitter and receiver can be terminated unexpectedly by the receiver. The connection termination timer implemented in machine R3 functions as expected, however the condition that resets this counter never occurs. The receiver only reset the timer when it receives a control packet from the transmitter. However, when SNR is operating in Mode 1, the transmitter never sends a control packet. The interaction expected by the receiver with the transmitter does not take place.

The problem detected in the SCM specification of the connection establishment phase [Tipi93], where the transmitter is ready to send data but the receiver has terminated the connection, is not considered serious. Even though the connection establishment phase seems to exhibit incorrect behavior, actions in the data transfer phase result in the transmitter also terminating the connection and all machines reset to their initial conditions.

The verification of SNR's data transfer phase operating with both error and flow control (Mode 2) was not completed due to difficulties encountered during the examination of Mode 1. These issues are discussed in the next section.

B. APPLYING MURPHI TO PROTOCOLS

Murphi was used successfully to verify properties of the connection establishment phase and the data transfer phase operating in Mode 0 and Mode 1. It appears possible but very difficult to apply Murphi to SNR's data transfer phase operating in Mode 2 and to the entire protocol (using a Murphi description that includes all phases and modes). Addressed below are issues related specifically to protocol verification with Murphi and limitations of Murphi in general.

The prevailing models used for protocols employ finite state machines and shared variables. For some asynchronous concurrent processes, the shared variables are relatively simple and easily implemented in Murphi's Descriptive Language. However for protocols, a network channel when included as one of the shared variables adds significantly to the complexity of the Murphi description. Three difficulties arises when implementing communication channel in Murphi.

1. Implementing the channel as an array of records (each array element is a slot for a packet and each of the record's field corresponds to a packet field) or a similar data structure adds a very large number of states to the state space. For example, a full description of SNR would requires a minimum channel length of four slots (two blocks of two packets) with each slot containing six fields. The domain of each field varies and depends on the actual values used in the description, however if roughly the same magnitude as used for the description in Chapter VII is assumed, then the number of states contributed by

the channels alone is approximately 7,000 states. Remember changing the value of any field of one of the channel slots changes the global state of the protocol being checked.

2. Real network channels are unreliable. They lose packets, corrupt data, and reorder packets. An accurate implementation must simulation network introduced errors.

3. Propagation delay is inherit in networks. The implementation should account for the time delay associated with the arrival of packet at their destination.

A clock mechanism to properly simulate the value of variable *clock_tick* was not required for the work done in this thesis. However, when all of SNR's machines are included in the Murphi description, it appears *clock_tick* will be required to accurately characterize SNR behavior. A practical implementation of a clock mechanism in Murphi should be developed and tested.

Once deadlock is reached on any execution path, verification halts and other paths are not checked. There is no simple method to ensure the first deadlock encountered is not masking another deadlocked path. Checking all paths for deadlock requires either the use of specific invariants coupled with disabling the detection of deadlock (a option of Murphi's special purpose verifier) or conditions causing deadlock must be corrected as they are detected. Under some conditions selecting a depth-first search strategy may uncover a deadlock different from one reached using a breadth-first search.

When an invariant fails, verification halts. If there are other invariants listed in the description after the one that failed, they are not tested. To check other invariants, the failing invariant must be removed and then the verification started again. This is really only an annoyance vice an actual limitation.

Overall Murphi is fairly easy to use. Producing an accurate Murphi description from a specification can be fairly challenging. (However translating a SCM specification, with its guarded transitions, into Murphi's descriptive language is straight forward.) The most difficult task is correctly expressing the desired invariants. Once this is done initial analysis can start immediately. Interpreting Murphi's output is not difficult, however as the number of states increases detecting implementation errors and identifying their source becomes extremely tedious.

C. FURTHER RESEARCH OPPORTUNITIES

1. SNR

The primary opportunity to expand upon the groundwork established with this thesis is to complete the verification of SNR. First the a single source specification must be written. The differences between the various documents describing SNR should be resolved and their content synthesized into a comprehensive specification. This master specification could then be analyzed and modified as design flaws are discovered. After modification each new version should be reanalyzed. The cycle should continue until the protocol exhibits the desired behavior. Specific behavior recommended to be checked include:

- Examine the situation where the receiver's buffer is full of partial blocks (i.e., blocks missing one or more packets). In this situation, none of the blocks will be acknowledged so retransmission is required. However since the buffer is full, retransmission can not occur. It appears deadlock will occur, does it?
- Does the protocol function properly when control packets containing erroneous information (corrupted by the channel) are encountered?.
- What happens if the values of T_{in} in the transmitter and T_{in} in the receiver differ significantly? Does an unexpected disconnect occurs?
- Investigate self stabilization in SNR. (If placed in an unsafe state, eventual the protocol reaches a safe state.) The originator of SNR claim SNR is self stabilizing in paragraph VII of [NRS90] . Is the periodic exchange of state information sufficient to recover from an unexpected condition, such as a momentary failure of T3?

2. Murphi

Two areas within the context of Murphi to explore further are: 1) support of communication channels and 2) using Murphi to investigate self stabilization.

It would be beneficial to eliminate network channels from the global state space and allow real network conditions to be generated. This could be accomplished by incorporating data channels as part of the underlying implementation of Murphi. It would allow the user to focus on the protocol being verified, vice the modeling and implementation of the network. The user would be reasonably sure that the channels are free of errors, and that any errors encountered were in the protocol under development. To permit the verification of various protocols the channels should be able to be tailored by the user. A channel implementation should include the following controllable parameters:

- Type of channel -- simplex, duplex, or multipaths -- between each node.
- Number of nodes comprising the network.
- Type of errors the channel could inject, such as data loss, garbling of data, lost packets, reordering of packets, unanticipated disconnection, etc.
- Error injection rate.
- Channel capacity and data rate.
- Prorogation delay.
- Type of network -- datagram or virtual circuit.

It would be interesting to explore further how an automatic verifier such as Murphi could be exploited for examining self stabilization of a concurrent system. Murphi can be used to determine is a system placed in an unsafe state reaches a safe state. The steps are:

- Generate the description for the concurrent system.
- Write an invariant for safe states.
- Negate the invariant so that when in an unsafe state the invariant is now true and is violated when a safe state is entered.
- Use the startstate construct to begin the verification process in an unsafe state.
- Run Murphi from an unsafe state and violation of the negation of the invariant will indicate when a safe state has been entered.

The problem comes in generating all possible unsafe state to be tested as start states.

APPENDIX A. DEADLOCK EXECUTION TRACE

```
===================================================================
Murphi Beta Release 2.73S (With Symmetry)
Finite-state Concurrent System Verifier.

Copyright (C) 1992, 1993
by the Board of Trustees of Leland Stanford Junior University.

===================================================================
This program should be regarded as a DEBUGGING aid, not as a
certifier of correctness.
Call with the -l flag or read the license file for terms
and conditions of use.
Run this program with "-h" for the list of options.

Bugs, questions, and comments should be directed to
"murphi@snooze.stanford.edu".

Murphi compiler last modified date: Apr 14 1994
Include files   last modified date: Apr 14 1994
===================================================================

Algorithm:
        Verification by breadth first search.
        with symmetry algorithm 1 -- fast canonicalization.

Memory usage:

        * The size of each state is 10 bits (rounded up to 2 bytes).
        * The memory allocated for the hash table is 2 Mbytes.
          With two words of overhead per state, the maximum size of
          the state space is 153871 states.
           * Use option "-k" or "-m" to increase this, if necessary.
        * Capacity in queue for breadth-first search: 38467 states.
           * Change the constant gPercentActiveStates in mu_verifier.h
             to increase this, if necessary.

===================================================================
Verbose option selected.  The following is the detailed progress.
```

```
Firing startstate Startstate 0
Obtained state:
P1:L1_1
P2:L2_1
C1 : 1
C2 : 1

----------------------------
Unpacking state from queue:
P1:L1_1
P2:L2_1
C1 : 1
C2 : 1

The following next states are obtained:

Firing rule 2P non-critical section
Obtained state:
P1:L1_1
P2:L2_2
C1 : 1
C2 : 1

Firing rule P1 non-critical section
Obtained state:
P1:L1_2
P2:L2_1
C1 : 1
C2 : 1

----------------------------
Unpacking state from queue:
P1:L1_1
P2:L2_2
C1 : 1
C2 : 1

The following next states are obtained:

Firing rule P2 assign C1 0
Obtained state:
P1:L1_1
P2:L2_3
C1 : 1
C2 : 0
```

```
Firing rule P1 non-critical section
Obtained state:
P1:L1_2
P2:L2_2
C1 : 1
C2 : 1

----------------------------
Unpacking state from queue:
P1:L1_2
P2:L2_1
C1 : 1
C2 : 1

The following next states are obtained:

Firing rule 2P non-critical section
Obtained state:
P1:L1_2
P2:L2_2
C1 : 1
C2 : 1

Firing rule P1 assign C1 0
Obtained state:
P1:L1_3
P2:L2_1
C1 : 0
C2 : 1

----------------------------
Unpacking state from queue:
P1:L1_1
P2:L2_3
C1 : 1
C2 : 0

The following next states are obtained:

Firing rule P2 wait
Obtained state:
P1:L1_1
P2:L2_4
C1 : 1
C2 : 0
```

```
Firing rule P1 non-critical section
Obtained state:
P1:L1_2
P2:L2_3
C1 : 1
C2 : 0

----------------------------
Unpacking state from queue:
P1:L1_2
P2:L2_2
C1 : 1
C2 : 1

The following next states are obtained:

Firing rule P2 assign C1 0
Obtained state:
P1:L1_2
P2:L2_3
C1 : 1
C2 : 0

Firing rule P1 assign C1 0
Obtained state:
P1:L1_3
P2:L2_2
C1 : 0
C2 : 1

----------------------------
Unpacking state from queue:
P1:L1_3
P2:L2_1
C1 : 0
C2 : 1

The following next states are obtained:

Firing rule 2P non-critical section
Obtained state:
P1:L1_3
P2:L2_2
C1 : 0
C2 : 1
```

```
Firing rule P1 wait
Obtained state:
P1:L1_4
P2:L2_1
C1 : 0
C2 : 1

-----------------------------
Unpacking state from queue:
P1:L1_1
P2:L2_4
C1 : 1
C2 : 0

The following next states are obtained:

Firing rule  critical section
Obtained state:
P1:L1_1
P2:L2_5
C1 : 1
C2 : 0

Firing rule P1 non-critical section
Obtained state:
P1:L1_2
P2:L2_4
C1 : 1
C2 : 0

-----------------------------
Unpacking state from queue:
P1:L1_2
P2:L2_3
C1 : 1
C2 : 0

The following next states are obtained:

Firing rule P2 wait
Obtained state:
P1:L1_2
P2:L2_4
C1 : 1
C2 : 0
```

```
Firing rule P1 assign C1 0
Obtained state:
P1:L1_3
P2:L2_3
C1 : 0
C2 : 0

----------------------------
Unpacking state from queue:
P1:L1_3
P2:L2_2
C1 : 0
C2 : 1

The following next states are obtained:

Firing rule P2 assign C1 0
Obtained state:
P1:L1_3
P2:L2_3
C1 : 0
C2 : 0

Firing rule P1 wait
Obtained state:
P1:L1_4
P2:L2_2
C1 : 0
C2 : 1

----------------------------
Unpacking state from queue:
P1:L1_4
P2:L2_1
C1 : 0
C2 : 1

The following next states are obtained:

Firing rule 2P non-critical section
Obtained state:
P1:L1_4
P2:L2_2
C1 : 0
C2 : 1
```

```
Firing rule  critical section
Obtained state:
P1:L1_5
P2:L2_1
C1 : 0
C2 : 1

-------------------------------
Unpacking state from queue:
P1:L1_1
P2:L2_5
C1 : 1
C2 : 0

The following next states are obtained:

Firing rule P2 assign C2 1
Obtained state:
P1:L1_1
P2:L2_1
C1 : 1
C2 : 1

Firing rule P1 non-critical section
Obtained state:
P1:L1_2
P2:L2_5
C1 : 1
C2 : 0

-------------------------------
Unpacking state from queue:
P1:L1_2
P2:L2_4
C1 : 1
C2 : 0

The following next states are obtained:

Firing rule  critical section
Obtained state:
P1:L1_2
P2:L2_5
C1 : 1
C2 : 0
```

```
Firing rule P1 assign C1 0
Obtained state:
P1:L1_3
P2:L2_4
C1 : 0
C2 : 0

-----------------------------
Unpacking state from queue:
P1:L1_3
P2:L2_3
C1 : 0
C2 : 0

The following next states are obtained:

Firing rule P2 wait
Obtained state:
P1:L1_3
P2:L2_3
C1 : 0
C2 : 0

Firing rule P1 wait
Obtained state:
P1:L1_3
P2:L2_3
C1 : 0
C2 : 0
==========================================================================

Result:
        Deadlocked state found.

State Space Explored:
        17 states, 26 rules fired in 0.40s.

Rules Information:
        Fired 1 times    - Rule "P2 assign C2 1"
        Fired 2 times    - Rule " critical section "
        Fired 3 times    - Rule "P2 wait"
        Fired 3 times    - Rule "P2 assign C1 0"
        Fired 4 times    - Rule "2P non-critical section"
        Fired 0 times    - Rule "P assign C 1"
        Fired 1 times    - Rule " critical section "
        Fired 3 times    - Rule "P1 wait"
        Fired 4 times    - Rule "P1 assign C1 0"
        Fired 5 times    - Rule "P1 non-critical section"
```

APPENDIX B. BUFFER OVERFLOW EXECUTION TRACE

```
==========================================================================
Murphi Beta Release 2.73S (With Symmetry)
Finite-state Concurrent System Verifier.
Copyright (C) 1992, 1993
by the Board of Trustees of Leland Stanford Junior University.
==========================================================================
This program should be regarded as a DEBUGGING aid, not as a
certifier of correctness.
Call with the -l flag or read the license file for terms
and conditions of use.
Bugs, questions, and comments should be directed to
"murphi@snooze.stanford.edu".
Murphi compiler last modified date: Apr 14 1994
Include files   last modified date: Apr 14 1994
==========================================================================
Algorithm:
        Verification by breadth first search.
        with symmetry algorithm 1 -- fast canonicalization.
Memory usage:
        * The size of each state is 83 bits (rounded up to 11 bytes).
        * The memory allocated for the hash table is 2 Mbytes.
          With two words of overhead per state, the maximum size of
          the state space is 95239 states.
           * Use option "-k" or "-m" to increase this, if necessary.
        * Capacity in queue for breadth-first search: 23809 states.
Progress Report:
        1000 states explored in 1.80s, with 2225 rules fired and 322 states in the queue.
        2000 states explored in 3.16s, with 4740 rules fired and 603 states in the queue.
        3000 states explored in 4.57s, with 7344 rules fired and 864 states in the queue.
        4000 states explored in 6.00s, with 9985 rules fired and 1113 states in the queue.
        5000 states explored in 7.49s, with 12746 rules fired and 1326 states in the queue.
        6000 states explored in 8.99s, with 15501 rules fired and 1536 states in the queue.
        7000 states explored in 10.37s, with 18025 rules fired and 1814 states in the queue.
        8000 states explored in 11.90s, with 20845 rules fired and 1997 states in the queue.
        9000 states explored in 13.32s, with 23451 rules fired and 2259 states in the queue.
        10000 states explored in 14.72s, with 26034 rules fired and 2513 states in the queue.
        11000 states explored in 16.23s, with 28816 rules fired and 2724 states in the queue.
        12000 states explored in 17.64s, with 31441 rules fired and 2989 states in the queue.
        13000 states explored in 19.04s, with 33996 rules fired and 3244 states in the queue.
        14000 states explored in 20.51s, with 36690 rules fired and 3489 states in the queue.
        15000 states explored in 22.01s, with 39495 rules fired and 3722 states in the queue.
        16000 states explored in 23.44s, with 42135 rules fired and 3983 states in the queue.
        17000 states explored in 24.84s, with 44713 rules fired and 4239 states in the queue.
        18000 states explored in 26.21s, with 47278 rules fired and 4552 states in the queue.
        19000 states explored in 27.72s, with 50072 rules fired and 4757 states in the queue.
```

The following is the error trace for the error:

Invariant -- no buffer overflow -- failed.

```
Startstate Startstate 0 fired.
T1_state:ts1
T2_state:ts4
R1_state:rs1
R3_state:rs1
T_CHAN[0].packet_kind:none_T
T_CHAN[1].packet_kind:none_T
R_CHAN[0].packet_kind:none_R
R_CHAN[0].buffer_avail : 2
R_CHAN[1].packet_kind:none_R
R_CHAN[1].buffer_avail : 2
xtmr_end_TC : 0
rcvr_end_TC : 0
xtmr_end_RC : 0
rcvr_end_RC : 0
k_T : 1
k_R : 1
latest_Tpacket.packet_kind:none_T
latest_Rpacket.packet_kind:none_R
latest_Rpacket.buffer_avail : 2
blk_seq_num : 0
OUTBUF : 3
buffer_avail : 2
buffer_avail_T : 2
UW_T : 0
LW_R : 0
LW_T : 0
T_busy : false
R_busy : false
scount_R : 0
count_R : 0
----------

Rule R3 - clock_tick -  rs1 fired.
R3_state:rs2
scount_R : 1
----------

Rule T1 - transmit possible - ts1 fired.
T1_state:ts4
----------

Rule T1 - transmit block - ts4 fired.
T1_state:ts1
T_CHAN[0].packet_kind:datapac
xtmr_end_TC : 1
blk_seq_num : 1
```

```
OUTBUF : 2
buffer_avail_T : 1
UW_T : 1
T_busy : true
----------

Rule R1 - receive data packet - rs1 fired.
R1_state:rs2
T_CHAN[0].packet_kind:none_T
rcvr_end_TC : 1
latest_Tpacket.packet_kind:datapac
R_busy : true
----------

Rule R3 - busy -  rs2 fired.
R3_state:rs4
----------

Rule R3 - send rcvr state -  rs4 fired.
R3_state:rs1
R_CHAN[0].packet_kind:conpac
rcvr_end_RC : 1
R_busy : false
----------

Rule R3 - clock_tick -  rs1 fired.
R3_state:rs2
scount_R : 2
----------

Rule R1 - process data packet -  rs2 fired.
R1_state:rs3
----------
Rule R1 - store data packet -  rs3 fired.
R1_state:rs1
buffer_avail : 1
----------

Rule T2 - receive rcvr state info - ts4 fired.
T2_state:ts5
R_CHAN[0].packet_kind:none_R
xtmr_end_RC : 1
latest_Rpacket.packet_kind:conpac
----------

Rule T2 - update info about rcvr - ts5 fired.
T2_state:ts6
buffer_avail_T : 2
----------

Rule T1 - transmit possible - ts1 fired.
```

```
T1_state:ts4
----------

Rule T1 - transmit block - ts4 fired.
T1_state:ts1
T_CHAN[1].packet_kind:datapac
xtmr_end_TC : 0
blk_seq_num : 2
OUTBUF : 1
buffer_avail_T : 1
UW_T : 2
----------

Rule R1 - receive data packet - rs1 fired.
R1_state:rs2
T_CHAN[1].packet_kind:none_T
rcvr_end_TC : 0
R_busy : true
----------

Rule R3 - busy -  rs2 fired.
R3_state:rs4
----------

Rule R3 - send rcvr state -  rs4 fired.
R3_state:rs1
R_CHAN[1].packet_kind:conpac
R_CHAN[1].buffer_avail : 1
rcvr_end_RC : 0
R_busy : false
----------

Rule R1 - process data packet -  rs2 fired.
R1_state:rs3
----------

Rule R1 - store data packet -  rs3 fired.
R1_state:rs1
buffer_avail : 0
----------

Rule T1 - transmit possible - ts1 fired.
T1_state:ts4
----------

Rule T1 - transmit block - ts4 fired.
T1_state:ts1
T_CHAN[0].packet_kind:datapac
xtmr_end_TC : 1
blk_seq_num : 3
OUTBUF : 0
```

```
buffer_avail_T : 0
UW_T : 3
----------
Rule R1 - receive data packet - rs1 fired.
R1_state:rs2
T_CHAN[0].packet_kind:none_T
rcvr_end_TC : 1
R_busy : true
----------

Rule R1 - process data packet -  rs2 fired.
R1_state:rs3
----------

Rule R1 - store data packet -  rs3 fired.
T1_state:ts1
T2_state:ts6
R1_state:rs1
R3_state:rs1
T_CHAN[0].packet_kind:none_T
T_CHAN[1].packet_kind:none_T
R_CHAN[0].packet_kind:none_R
R_CHAN[0].buffer_avail : 2
R_CHAN[1].packet_kind:conpac
R_CHAN[1].buffer_avail : 1
xtmr_end_TC : 1
rcvr_end_TC : 1
xtmr_end_RC : 1
rcvr_end_RC : 0
k_T : 1
k_R : 1
latest_Tpacket.packet_kind:datapac
latest_Rpacket.packet_kind:conpac
latest_Rpacket.buffer_avail : 2
blk_seq_num : 3
OUTBUF : 0
buffer_avail : -1
buffer_avail_T : 0
UW_T : 3
LW_R : 0
LW_T : 0
T_busy : true
R_busy : true
scount_R : 2
count_R : 0
----------

End of the error trace.
```

Result:

Invariant -- no buffer overflow -- failed.

State Space Explored:

19652 states, 51943 rules fired in 28.81s.

Rules Information:

```
Fired 0 times     - Rule "R3 - disconnect -  rs4"
Fired 1904 times - Rule "R3 - send rcvr state -  rs4"
Fired 1564 times - Rule "R3 - modify k_R -  rs3"
Fired 1556 times - Rule "R3 - wait (count_R < k_R) -  rs3"
Fired 1126 times - Rule "R3 - busy -  rs2"
Fired 3033 times - Rule "R3 - not busy -  rs2"
Fired 5557 times - Rule "R3 - clock_tick -  rs1"
Fired 5900 times - Rule "remove packet from buffer"
Fired 3229 times - Rule "R1 - store data packet -  rs3"
Fired 4292 times - Rule "R1 - process data packet -  rs2"
Fired 2116 times - Rule "R1 - receive data packet - rs1"
Fired 5746 times - Rule "T2 - go back to ts4 - ts6"
Fired 2914 times - Rule "T2 - update info about rcvr - ts5"
Fired 3714 times - Rule "T2 - receive rcvr state info - ts4"
Fired 3534 times - Rule "T1 - transmit block - ts4"
Fired 5758 times - Rule "T1 - transmit possible - ts1"
```

LIST OF REFERENCES

[Ben82] Ben-Ari, M., *Principles of Concurrent and Distributed Programming,* Prentice- Hall, 1982.

[Ben90] Ben-Ari, M., *Principles of Concurrent and Distributed Programming,* Prentice Hall, 1990.

[Ben93] Ben-Ari, M., *Mathematical Logic for Computer Science*, Prentice Hall, 1993.

[ChHa92] Chow, J., and Harrison, W., "Compile-time Analysis of Parallel Programs that Share Memory", *ACM 19th Symposium on Principles of Programming Languages,* 1992.

[ChHa94] Chow, J., and Harrison, W., "State Space Reduction in Abstract Interpretation of Parallel Programs" IEEE 1074-8970/94 1994.

[CGL92] Clarke, E., Grumberg, O., and Long, D., "Model Checking and Abstraction", *ACM 19th Symposium on Principles of Programming Languages*, 1992.

[DDHY92] Dill, D., Drexler, A., Hu, A., and Yang, C. H., "Protocol Verification as a Hardware Design Aid", *IEEE International Conference on Computer Design*, October 1992.

[GoMu91] Gouda, M., and Multari, N., "Stabilizing Communication Protocols", *IEEE Transaction on Computers,* Vol. 40. No 4., April 1991.

[Gri95] Grier, R., "Testing an Implementation's Conformance to a Formal Specification: the SNR High-Speed Transport Protocol", M.S. Thesis, Naval Postgraduate School, Monterey, CA., March 1995.

[IpD93] Ip, C. N., and Dill, D., "Better Verification Through Symmetry", *Proceedings of the 11th International Symposium on Computer Hardware Description Language and Their Application,*, Cambridge, MA, October 1993.

[IpDi93] Ip, C. N., and Dill, D., "Efficient Verification of Symmetric Concurrent Systems", *IEEE International Conference on Computer Design: VLSI in Computers and Processors,* Cambridge, MA, October 1993.

[LuTi94] Lundy, G., and Tipici, H., "Specification and Analysis of the SNR High-Speed Transport Protocol", *IEEE/ACM Transactions on Networking*, Vol. 2, No. 5, October 1994.

[McAr92] McArthur, R., ., "Design and Specification of a High-Speed Transport Protocol", M.S. Thesis, Naval Postgraduate School, Monterey, CA., March 1992.

[McMi92] McMillan, K., *The SMV System DRAFT*, Carnegie-Mellon University, February 1992.

[MeDi93] Ralph Melton, and R., Dill, D., "Murphi Annotated Reference Manual", Version 2.73S, Stanford University, November 1993.

[Mez95] Mezhound, F., "Implementation of the SNR High-Speed Transport Protocol (the Transmitter Part)", M.S. Thesis, Naval Postgraduate School, Monterey, CA., March 1995.

[NRS90] Netravali, A., Roome, W., and Sabnani, K., "Design and Implementation of a High Speed Transport Protocol", *IEEE Transactions in Communications*, Vol. 38, No. 11, November 1990.

[Tipi93] Tipici, H., "Specification and Analysis of a High Speed Transport Protocol", M.S. Thesis, Naval Postgraduate School, Monterey, CA., June 1993.

[Wan95] Wan, W., "Implementation of the SNR High-Speed Transport Protocol (the Receiver Part)", M.S. Thesis, Naval Postgraduate School, Monterey, CA., March 1995.

[Zha86] Zhang, L., "Why TCP Timers Don't Work Well", *Proceedings of the ACM SIGCOM Symposium on Communications, Architectures, and Protocols*, February 1986.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center.. 2
   Cameron Station
   Alexandria, Virginia 22304-6145

2. Library, Code 52... 2
   Naval Postgraduate School
   Monterey, California 93943-5101

3. Chairman, Code CS.. 1
   Computer Science Department
   Naval Postgraduate School
   Monterey, California 93943

4. Dr. D. Volpano, Code CS/Vo... 2
   Computer Science Department
   Naval Postgraduate School
   Monterey, California 93943

5. Dr. G.M. Lundy, Code CS/Lu... 2
   Computer Science Department
   Naval Postgraduate School
   Monterey, California 93943

6. CDR Carl M Pederson, Jr., USN 1
   PSC 78 BOX 346
   APO AP 96326-0346

7. Krishan K. Sabnani... 1
   AT&T Bell Laboratories
   Room 4G - 502
   101 Crawford's Corner Road
   Holmdel, New Jersey 07733